8-24-1992

Issues in the design and implementation of a real-time garbage collection architecture

William J. Schmidt
Iowa State University

Issues in the design and implementation of a real-time garbage collection architecture

Abstract
This dissertation proposes a new garbage-collected memory module architecture for hard real-time systems. The memory module is designed for compatibility with standard workstation architectures, and cooperates with standard cache consistency protocols. Processes read and write garbage- collected memory in the same manner as standard memory, with identical performance under most conditions. Occasional contention between user processes and the garbage collector results in delays to the user process of at most six memory cycles. Thus the proposed architecture guarantees real-time performance at fine granularity. This dissertation investigates the viability of the proposed architecture in two senses. First, it demonstrates that a fundamental component of the architecture, the object space manager, can be produced at a reasonable cost. Second, this dissertation reports the results of experiments that measure the performance of the proposed architecture under real workloads. Results of these experiments show that the architecture currently performs more slowly than traditional schemes; but this appears to be correctable by employing a more efficient function call mechanism that caches heap- allocated activation frames. Finally, this dissertation reports on some simple extensions to the C++ programming language to support slice objects. Slice objects, which are supported by the garbage collection architecture, are useful for implementing fragmentable arrays, i.e., arrays in which subarrays may be retained while unused elements become garbage and are collected. Experimental evidence demonstrates that slice objects can be used to implement strings more efficiently than at least some popular class libraries.

Disciplines
Systems Architecture | Theory and Algorithms

Issues in the design and implementation of a real-time garbage collection architecture

TR 92-25
William Jon Schmidt

August 24, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

Issues in the design and implementation of a real-time garbage collection architecture

by

William Jon Schmidt

An Abstract of

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa
1992

Issues in the design and implementation of a real-time garbage collection architecture

William Jon Schmidt

Under the supervision of Dr. Kelvin D. Nilsen
From the Department of Computer Science
Iowa State University

This dissertation proposes a new garbage-collected memory module architecture for hard real-time systems. The memory module is designed for compatibility with standard workstation architectures, and cooperates with standard cache consistency protocols. Processes read and write garbage-collected memory in the same manner as standard memory, with identical performance under most conditions. Occasional contention between user processes and the garbage collector results in delays to the user process of at most six memory cycles. Thus the proposed architecture guarantees real-time performance at fine granularity.

This dissertation investigates the viability of the proposed architecture in two senses. First, it demonstrates that a fundamental component of the architecture, the *object space manager*, can be produced at a reasonable cost. Second, this dissertation reports the results of experiments that measure the performance of the proposed architecture under real workloads. Results of these experiments show that the architecture currently performs more slowly than traditional schemes; but this appears to be correctable by employing a more efficient function call mechanism that caches heap-allocated activation frames.

Finally, this dissertation reports on some simple extensions to the C++ programming language to support *slice objects*. Slice objects, which are supported by the garbage collection architecture, are useful for implementing *fragmentable arrays*, i.e., arrays in which subarrays may be retained while unused elements become garbage and are collected. Experimental evidence demonstrates that slice objects can be used to implement strings more efficiently than at least some popular class libraries.[1]

[1]This research was supported in part by NSF Grant MIP-9010412 and by a National Science Foundation Graduate Fellowship.

Issues in the design and implementation of a real-time garbage collection architecture

by

William Jon Schmidt

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Members of the Committee:

Iowa State University
Ames, Iowa
1992

TABLE OF CONTENTS

1. INTRODUCTION, HISTORY, AND MOTIVATION 1
1.1 Problem statement 1
1.2 Previous work in this area 4
2. THE ISU REAL-TIME GARBAGE COLLECTION PROJECT 8
2.1 The real-time garbage-collection algorithm 9
2.2 Object types 12
2.3 The garbage-collected memory module architecture 13
2.3.1 Overall system architecture 13
2.3.2 Motivation for the object-space manager 16
2.3.3 Alternatives to the object space manager 17
3. THE OBJECT SPACE MANAGER 19
3.1 Design criteria 19
3.2 Interconnection architecture and command interface 20
3.3 Some design details 24
3.3.1 Command set implementation 28
3.4 Analysis 31
3.4.1 VLSI technologies 31
3.4.2 Transistor counts 32
3.4.3 Wire costs 36
3.4.4 Propagation delays 37
3.5 Improving performance 39
3.6 Other alternatives 42
3.7 Conclusions and future work 45

4. A PROTOTYPE COMPILER IMPLEMENTATION 47
4.1 The C++ compiler 48
4.1.1 The arbiter interface 48
4.1.2 The virtual machine 52
4.1.3 Pointer location descriptions 55
4.1.4 Special data objects 60
4.1.5 The run-time library 63
4.1.6 Optimizations 64
4.1.7 Compatibility considerations 66
4.1.8 Limitations 69
4.2 The linker and librarian 70
4.3 The `dlxsimgc` simulator 72
4.3.1 Explanation of statistics 73
4.3.2 Limitations 76
5. PERFORMANCE ANALYSIS 78
5.1 System definitions 78
5.2 Parameters and factors 80
5.2.1 System parameters 80
5.2.2 Workload parameters 81
5.3 Workload 82
5.4 Results of experiments 84
5.4.1 Elapsed CPU cycles 87
5.4.2 Execution latencies 90
5.4.3 CPU instructions executed 92
5.4.4 Allocation latencies 96
5.4.5 Cache performance 99
5.4.6 Difference in costs and latencies 113
5.4.7 Allocations impeded by garbage collection 118
5.4.8 Cycles required for garbage collection 118
5.4.9 Fraction of time that garbage collection is active 119
5.4.10 Bus utilization 122
5.4.11 Bus utilization due to cache invalidation 125

5.4.12 Additional statistics 125
5.5 Additional experiments 127
5.5.1 Partial cache invalidation 127
5.5.2 The effects of garbage-collected memory size 132
5.6 Summary 136
6. ALTERNATIVE FUNCTION CALL MECHANISMS 138
6.1 Implementation 139
6.1.1 Compiling function calls using shared argument blocks 148
6.1.2 Compiling function calls using separate argument blocks . . . 149
6.2 Results of experiments 152
6.3 A solution to the stack manipulation overhead problem 162
6.3.1 Performance model of activation frame caching 163
6.3.2 Preliminary experimental results 170
6.3.3 A new approach to data coherence 174
7. LANGUAGE EXTENSIONS TO SUPPORT SLICE OBJECTS 177
7.1 Syntax and informal semantics 178
7.1.1 Declarations 179
7.1.2 Expressions 179
7.1.3 Possible extensions 181
7.2 Implementation notes 182
7.3 Results of experiments 185
8. CONCLUSIONS AND FUTURE WORK 189
ACKNOWLEDGMENTS 191
BIBLIOGRAPHY 193

LIST OF TABLES

Table 3.1: Transistor cost parameters 33
Table 3.2: Transistor costs by technology 35
Table 3.3: Object space per OSM chip, given chip density and object size 37
Table 3.4: Ratio of OSM chips to DRAM chips 37

Table 4.1: Register usage in the two C++ compilers 54

Table 5.1: Elapsed CPU cycles, `sfft` 88
Table 5.2: Elapsed CPU cycles, `lisp` 88
Table 5.3: Elapsed CPU cycles, `troff` 89
Table 5.4: Elapsed CPU cycles, all experiments 89
Table 5.5: Total latencies, `sfft` (normal CPU cycles) 90
Table 5.6: Total latencies, `lisp` (normal CPU cycles) 91
Table 5.7: Total latencies, `troff` (normal CPU cycles) 91
Table 5.8: Total latencies, all experiments (normal CPU cycles) 92
Table 5.9: Total instructions executed, `sfft` 93
Table 5.10: Total instructions executed, `lisp` 93
Table 5.11: Total instructions executed, `troff` 94
Table 5.12: Number of instructions executed, all experiments 94
Table 5.13: Breakdown of arbiter calls 95
Table 5.14: Cost of stack manipulation 96
Table 5.15: Total latencies for allocations, `sfft` 97
Table 5.16: Total latencies for allocations, `lisp` 97
Table 5.17: Total latencies for allocations, `troff` 98
Table 5.18: Total latencies for allocations, all experiments 98
Table 5.19: Instruction cache hits, `sfft` 100

Table 5.20: Instruction cache fetches, `sfft` . . . 101
Table 5.21: Instruction cache hit rate, `sfft` . . . 101
Table 5.22: Instruction cache hits, `lisp` . . . 102
Table 5.23: Instruction cache fetches, `lisp` . . . 102
Table 5.24: Instruction cache hit rate, `lisp` . . . 103
Table 5.25: Instruction cache hits, `troff` . . . 103
Table 5.26: Instruction cache fetches, `troff` . . . 104
Table 5.27: Instruction cache hit rate, `troff` . . . 104
Table 5.28: Instruction cache hits, all experiments . . . 105
Table 5.29: Instruction cache fetches, all experiments . . . 105
Table 5.30: Instruction cache hit rate, all experiments . . . 106
Table 5.31: Data cache hits, `sfft` . . . 107
Table 5.32: Data cache fetches, `sfft` . . . 108
Table 5.33: Data cache hit rate, `sfft` . . . 108
Table 5.34: Data cache hits, `lisp` . . . 109
Table 5.35: Data cache fetches, `lisp` . . . 109
Table 5.36: Data cache hit rate, `lisp` . . . 110
Table 5.37: Data cache hits, `troff` . . . 110
Table 5.38: Data cache fetches, `troff` . . . 111
Table 5.39: Data cache hit rate, `troff` . . . 111
Table 5.40: Data cache hits, all experiments . . . 112
Table 5.41: Data cache fetches, all experiments . . . 112
Table 5.42: Data cache hit rate, all experiments . . . 113
Table 5.43: Total cycles difference between costs and latencies, `sfft` . . . 115
Table 5.44: Total cycles difference between costs and latencies, `lisp` . . . 115
Table 5.45: Total cycles difference between costs and latencies, `troff` . . . 115
Table 5.46: Total cycles difference between costs and latencies, all experiments . . . 116
Table 5.47: Wasted cycles for the standard GC configuration . . . 116
Table 5.48: Fraction of time wasted by protocol, `sfft` . . . 116
Table 5.49: Fraction of time wasted by protocol, `lisp` . . . 117
Table 5.50: Fraction of time wasted by protocol, `troff` . . . 117

Table 5.51: Fraction of time wasted by protocol, all experiments 117
Table 5.52: Number of allocations impeded by GC, all experiments . . . 119
Table 5.53: Total cycles required for GC, `sfft` 120
Table 5.54: Total cycles required for GC, `lisp` 120
Table 5.55: Total cycles required for GC, `troff` 120
Table 5.56: Total cycles required for GC, all experiments 121
Table 5.57: Fraction of time GC is active, `sfft` 121
Table 5.58: Fraction of time GC is active, `lisp` 121
Table 5.59: Fraction of time GC is active, `troff` 122
Table 5.60: Fraction of time GC is active, all experiments 122
Table 5.61: Bus utilization, `sfft` 123
Table 5.62: Bus utilization, `lisp` 123
Table 5.63: Bus utilization, `troff` 124
Table 5.64: Bus utilization, all experiments 124
Table 5.65: Bus utilization due to cache invalidation, `sfft` 125
Table 5.66: Bus utilization due to cache invalidation, `lisp` 126
Table 5.67: Bus utilization due to cache invalidation, `troff` 126
Table 5.68: Bus utilization due to cache invalidation, all experiments . . 126
Table 5.69: Effect of partial cache invalidation, `sfft/small` 129
Table 5.70: Effect of partial cache invalidation, `lisp/prune` 130
Table 5.71: Effect of partial cache invalidation, `troff/osmpaper` 131
Table 5.72: Effect of garbage-collected memory size 133

Table 6.1: Characteristics of alternative compilers 139
Table 6.2: Comparative compiler performance, `sfft` 156
Table 6.3: Comparative compiler performance, `lisp` 157
Table 6.4: Comparative compiler performance, combined workload . . . 158
Table 6.5: Expected overhead of proposed function call mechanism . . . 168
Table 6.6: Improvement of proposed mechanism over SU compiler . . . 169
Table 6.7: Results of HC experiment, `sfft/small` 171
Table 6.8: Results of HC experiment, `lisp/prune` 172
Table 6.9: Activation frame hit rates 172

Table 7.1: Examples of slice declarations 180
Table 7.2: Expression syntax for slice operations 180
Table 7.3: Internal representation for slice operations 184
Table 7.4: Results of `editor` experiments (1 of 2) 186
Table 7.5: Results of `editor` experiments (2 of 2) 188

LIST OF FIGURES

Figure 2.1: Organization of *to-space* 11
Figure 2.2: System architecture 14
Figure 2.3: Garbage-collected memory module internal architecture . . . 15

Figure 3.1: OSM interconnection 21
Figure 3.2: OSM pinouts 23
Figure 3.3: OSM structure 25
Figure 3.4: Control unit 26
Figure 3.5: Logic diagram for access tree node at level k 28
Figure 3.6: Typical slice of the OHB register interface 29

Figure 4.1: Arbiter ports 49
Figure 4.2: PLD structure 55
Figure 4.3: Example of PLD assembly code 56
Figure 4.4: Example of union member assignment 57
Figure 4.5: Syntax tree for lvalue of union assignment 58
Figure 4.6: Example requiring a dynamic PLD 59
Figure 4.7: Function call protocol 61
Figure 4.8: Incompatibility arising from `operator new` 67
Figure 4.9: A workaround for the `operator new` problem 68

Figure 6.1: Activation frame for the base compiler 140
Figure 6.2: Activation frame for the SS and SU compilers 141
Figure 6.3: Activation frame for the HS and HU compilers 142
Figure 6.4: Function prologue and epilogue code for base compiler 143
Figure 6.5: Function prologue and epilogue code for SS compiler 144

Figure 6.6: Function prologue and epilogue code for SU compiler 145
Figure 6.7: Function prologue and epilogue code for HS compiler 146
Figure 6.8: Function prologue and epilogue code for HU compiler 147
Figure 6.9: Example source causing excess argument pointer adjustments 153
Figure 6.10: Before peephole optimization 154
Figure 6.11: After peephole optimization 155

Figure 7.1: Example slice code 181

1. INTRODUCTION, HISTORY, AND MOTIVATION

1.1 Problem statement

The topic of this dissertation lies within the much broader area of language and architectural support for hard real-time systems. A *real-time* system is a computer system that is designed to perform under timing constraints determined by external and internal *events*. A *hard* real-time system is a real-time system in which the failure of a software or hardware component to meet its timing constraints can result in the incorrect operation of the system as a whole.

Although real-time computer systems have been with us for several decades, the support and study of such systems as a separate discipline within computer science has only recently evolved. To this day, real-time systems are designed primarily with ad hoc techniques. Designers of hard real-time systems are unwilling to use architectural features (such as caches) that exhibit better average performance at the expense of predictability, since a *single* failure to meet a deadline could, for example, scuttle an expensive deep-space mission.

As greater demands are placed upon future real-time systems, however, it will no longer be possible to rely upon ad hoc techniques, nor will it be possible to settle for low performance. Stankovic [47] has given the following characterization of these "next-generation" real-time systems:

> The next-generation real-time systems will be in similar application areas as current systems, but will be more complex in that they will be distributed, contain highly dynamic and adaptive behavior, exhibit intelligent behavior, have long lifetimes, and be characterized as having catastrophic consequences if the logical or timing constraints of the system are not met.

It is obvious that next-generation real-time systems will require the support of language facilities at least equal to those of time-independent computer systems. In reality, real-time language facilities and support tools lag far behind the rest of the industry.

Among the programming tools avoided by contemporary real-time systems developers is support for dynamic memory management. This is due to the unpredictable delays associated with most standard memory management techniques. Such delays are generally *state-dependent*, in the sense that they are determined by the sequence and sizes of allocations and deallocations that have previously been performed. This is true both of *explicit* storage allocation techniques (such as the use of malloc and free in the C language) and of the *implicit* garbage-collection and compaction algorithms found predominantly in symbolic languages. To provide guarantees of schedulable performance, current real-time systems usually avoid dynamic memory management as much as possible, often at the expense of wasted resources and developer effort.

If, however, the next-generation real-time systems are to "contain highly dynamic and adaptive behavior" and to "exhibit intelligent behavior," it will be impossible to avoid the use of dynamic memory management in the future. This leads to a choice between explicit storage management and garbage collection. In general, garbage collection is more appropriate for building reliable software systems. Although both methods allow unnecessary storage to be accidentally retained through careless programming, only explicit storage management allows the possibility of deleting an object and later attempting to reaccess it through a dangling reference. Furthermore, garbage collection facilitates production of correct programs by removing from the programmer the burden of explicitly deleting storage. Garbage collection will be crucial if the need for large, dynamic, reliably engineered real-time systems is to be satisfied.

If garbage collection is to be successfully incorporated into real-time systems, it must be adapted to suit these systems' special needs. Furthermore, this adaptation must be performed with an eye to the requirements of next-generation systems. The fundamental requirements of a garbage collector for real-time systems of the future are the following:

1. The worst-case latency associated with an allocation request must be short and

predictable.

2. The garbage collector must have a minimal impact on system efficiency.

3. The garbage collector must support allocation of space for every type of object needed in the system.

Requirement 3 may seem on the surface to be too obvious to mention, but in fact it places some added burdens on the designer. Early dialects of Lisp, for instance, were very easy to write garbage collectors for, since objects were of a fixed size and contained pointers to other objects only in fixed locations. In a language such as C++ that provides extensible typing, objects theoretically may be of any size and may contain pointers to other objects anywhere within them. From the point of view of the garbage collector designer, this is much more difficult to deal with. However, extensible typing is an inseparable part of data abstraction, which must certainly be provided for languages supporting next-generation real-time systems development. Furthermore, real-time programmers often use large contiguous objects rather than linked data structures to represent complex data, since this technique permits constant-time data access. Thus it is impractical to set artificial limits on the size of objects in a real-time garbage collector.

The purpose of the research described in this dissertation is to investigate the *practicality* of providing such "type-complete" garbage collection for use in real-time systems. The basis for this work is the recently-developed algorithm and architecture described in references [40, 41] and briefly discussed in chapter 2. The garbage-collection algorithm described in this reference is the first to sufficiently address requirement 3, above, within the context of real-time systems. Subsequent chapters of this dissertation discuss the feasibility of required hardware components, development of supporting compiler technology, and results of experiments to (i) determine the efficacy of the proposed garbage-collection system as a whole; (ii) provide statistics on garbage-collection behavior; (iii) investigate tradeoffs among alternative function call mechanisms; and (iv) measure the costs and benefits of "slice objects," which permit programmers to define fragmentable arrays in which unused elements are automatically reclaimed.

1.2 Previous work in this area

The concept of garbage collection in real time dates back at least to 1968, when Knuth [22] credited Minsky with the solution to the following exercise:

> Show that it is possible to use a garbage collection method reliably in a "real time" application, e.g., when a computer is controlling some physical device, even when stringent upper bounds are placed on the maximum execution time required for each List operation performed.

The solution sketched by Knuth was extended to multiprocessing by Steele [49], and the feasibility of this method was analyzed by Muller [35] and Wadler [55]. The algorithm described is a mark-and-sweep compacting collector, requiring three passes over heap storage for each collection. The analysis of the algorithm is quite difficult, and the availability of sufficient memory for an application is only guaranteed for an "equilibrium" condition in which the rates of marking, sweeping, and relocation, relative to cell allocation, are all known. Another multiprocessing algorithm was developed by Dijkstra, Lamport, and others [7, 8, 24, 25], but was reported by Baker [2] to be too inefficient for practical use, being designed primarily to support a proof that exactly those cells that are garbage are collected.

The first "successful" real-time garbage collection algorithm, providing upper bounds on both allocation latency and required storage (as a function of reachable storage), is the algorithm of Baker [2]. Baker's algorithm is based on the *copying* collection algorithm that was first introduced by Minsky [33]. The basic idea of the algorithm is to divide available memory into two large regions named *to-* and *from-space* respectively. Objects are allocated from *to-space* while previously allocated live objects are incrementally copied into *to-space* out of *from-space*. When there is no longer adequate memory in *to-space* to satisfy an allocation request, garbage collection begins. The names assigned to the two memory regions are exchanged, so that allocations are now made from the other region. This is called a *flip*. The design of the algorithm guarantees that all live data will have been copied out of the old *from-space* by the time the next flip occurs.

Baker's algorithm was developed for a Lisp system, and therefore was only concerned with a limited number of objects: CONS cells and "compact list representa-

tion" cells. These are, conveniently, all the same size, making it easy to copy objects atomically and guarantee tight upper bounds on performance. Baker also sketched how his system could be extended to allocate and collect vectors of raw (i.e., non-pointer) data, using methods he attributes to Steele [50]. However, Baker's algorithm was not yet sufficiently well-developed to be used for type-complete garbage collection.

As a first step in this direction, Nilsen [36] extended Baker's algorithm to the general problem of garbage collecting string data, as well as Lisp-like linked data structures, in real time. This algorithm is useful for many languages, such as Icon [14] and SNOBOL4 [15], that specifically support a string data type. Strings in such languages are often shared between numerous pointers, which may access different portions of a single string. During processing, it is often the case that only substrings of previously allocated string data remain accessible to user pointers. The remaining portions of the string data constitute garbage and may be collected. It is noteworthy that extending Baker's algorithm to include this single additional data type requires significant modification.

In comparison with stop-and-wait garbage collectors, the real-time garbage collectors discussed above generally perform very poorly. Users of the original implementation of Baker's real-time garbage collector found it to be so slow that they disabled it [10]. And Nilsen's real-time implementation of Icon runs two to three times slower than the traditional implementation, which uses mark-and-sweep garbage collection with compaction. Comparison of the two Icon implementations reveals that the real-time Icon implementation runs slower than traditional Icon by this same factor of two to three even with programs for which the burden of garbage collection is especially light. This observation, and detailed profiling of the real-time Icon interpreter, lead to the conclusion that the major cost of the real-time garbage collector is the overhead imposed on every memory read and write operation, not the time spent copying live data into *to-space*.

This conclusion is further corroborated by recent research described by Ellis, Li, and Appel in reference [10]. In this research, stock memory management hardware was used to reduce the software costs associated with each memory operation. Pages

of memory in *to-space* that have not yet been scanned[1] are flagged so that page faults interrupt execution whenever attempts are made to reference unscanned or uncopied data. Using this technique, the performance of the real-time algorithm is comparable in throughput with more traditional stop-and-wait garbage collectors. The time required to perform a flip is approximately 100 msec. The average time required to read from an unscanned page is approximately 50 msec.

These results validate the long-held consensus that copying collectors should exhibit superior performance in comparison with other well-known storage management schemes. In theory, the family of copying garbage collection algorithms is much more efficient than reference counting, explicit allocation and deallocation (malloc and free), and mark-and-sweep garbage collection [1, 26]. This is because the work performed by the garbage collector is proportional to the amount of live data at the time garbage collection takes place. This cost can be made arbitrarily small in relation to total storage throughput by increasing the sizes of *to-* and *from-space* appropriately. In comparison, the cost of reference counting is proportional to the number of memory operations that overwrite a pointer. The costs of explicit deallocation are proportional to the amount of data freed. And the cost of mark-and-sweep garbage collection is proportional to the total amount of data previously allocated.[2]

[1] *Scanning* refers to the process of updating *from-space* pointers within an object to point to new *to-space* copies of the referenced objects. See section 2.1 for more details.

[2] The superiority of copying collectors over mark-and-sweep collectors is a topic of recent dispute. Zorn [59] has obtained results showing that mark-and-sweep collectors need not be as comparatively inefficient as previously supposed, particularly when the effect of virtual memory paging on collector performance is observed. However, Wilson [58] has pointed out that modifying copying collectors to use generational scavenging techniques [30, 54] negates their poor paging performance. It appears that copying collectors still outperform mark-and-sweep collectors, but not by as much as had previously been supposed. Paging performance is not an issue for the garbage collection system described in this dissertation, since it uses "real" rather than virtual memory addressing. Real memory addressing is a common feature in hard real-time systems because of the unpredictable latencies produced by virtual memory schemes.

The findings presented in the Ellis, Li, Appel paper demonstrate that a copying garbage collector is capable of providing high throughput and bounded response time by using hardware to detect and handle those memory accesses that require special handling. However, the time required to flip and the worst-case time required to read a single word of memory are much too high to support important real-time applications such as robotics, radar signal processing, flight control of aircraft, and interactive multi-media workstations. To obtain faster response times, finer granularity of atomic actions is necessary.

To summarize, efforts prior to 1990 have shown that real-time garbage collection is feasible, but that more work is necessary to make it practical for use in real systems. The following avenues of inquiry are of primary interest:

1. Algorithms must be developed that support garbage collection for objects of any possible type in a type-extensible language.

2. Performance must be significantly enhanced.

Recent progress [40, 48] has been made towards a solution to item 1. The algorithm described in chapter 2 of this dissertation, and explained in more detail in reference [40], supports objects of any size and type, provided that the locations of all pointers are made known to the collector. To address item 2, a new garbage-collected memory architecture has been proposed. The overall architecture is discussed briefly in chapter 2, and at length in references [40, 41]. One design for the object space manager, a critical component of the hardware architecture, is described in chapter 3 of this dissertation. A more cost-effective alternative is outlined in reference [39].

Software support for the garbage-collection architecture includes a C++ compiler, linker, librarian, and simulator. The design and implementation of these tools is discussed in chapter 4. Using these tools, a number of experiments have been carried out to determine the efficacy of the proposed garbage-collection architecture. Chapters 5, 6, and 7 describe the design and results of these experiments. Chapter 8 summarizes conclusions from these efforts.

2. THE ISU REAL-TIME GARBAGE COLLECTION PROJECT

Since 1988, the problem of garbage collection in real time has been under study at Iowa State University under the direction of Dr. Kelvin Nilsen. This project is an outgrowth of Professor Nilsen's doctoral investigations into real-time garbage collection of Icon strings and linked data structures [36], described in section 1.2 above. The goal of this effort is to produce a garbage-collection architecture suitable for use in any real-time system, including those with very strict latency requirements, and capable of supporting any modern programming language.

A number of students have been involved in this project. Stapleton [48] made the first attempt to extend the Baker/Nilsen garbage-collection algorithm to include descriptor slice objects,[1] and was the first to discover the need for the object space manager investigated in chapter 3 below. Singh [45] designed an early prototype of the memory arbiter, another important component of the garbage-collection architecture described in section 2.3. These efforts contributed to a better understanding of pertinent design issues. Building upon this work, Nilsen and Schmidt [39, 40, 41] have designed a combination of hardware and software to address the problem of real-time garbage collection. The remainder of this chapter contains a brief overview of the garbage-collection algorithm and supporting hardware. Later chapters in this dissertation contain experimental results quantifying the successes and limitations of this design.

[1] A descriptor slice object is essentially an array object that contains pointers, with the property that *portions* of the object (rather than always the whole object) may become garbage. See section 2.2 for a more complete description.

2.1 The real-time garbage-collection algorithm

Garbage collection in real time is made possible by distributing the efforts of garbage collection over time. The algorithm is real-time in the following sense:

- Each memory allocation is accompanied by an amount of garbage collection that is proportional to the size of the allocation. The proportionality constant that relates garbage collection to memory allocation is defined when the garbage collector is configured. Based on this constant, real-time programmers are able to derive upper bounds on the time required to perform particular allocations.

- The allocation routine is interruptible, so high priority processes are never impeded by low-priority processes requesting allocation of very large objects.

- Based on the amount of physical memory available to the garbage collector and on the proportionality constant described above, it is straightforward to derive a guaranteed lower bound on the amount of memory that is always available for representation of live objects. This assures programmers of safety-critical real-time applications that their programs are not vulnerable to failure due to lack of memory for new allocations.

A thorough description of the algorithm and its analysis is provided in reference [40].

Recall that new objects are allocated from *to-space* while old objects are copied from *from-space* into *to-space*, with a flip (exchange of spaces) occurring when *to-space* has been exhausted. The application program is allowed to maintain only a limited number of pointers (called descriptors) to dynamically allocated objects. The descriptors under direct control of the application are called tended descriptors. When a flip occurs, the objects directly referenced by tended descriptors are scheduled for copying into *to-space*, and the descriptors are modified to reflect the new locations of the objects they refer to. The task of updating a pointer to reflect the new location of a live data object is called *tending*. The garbage collector maintains the invariant that tended descriptors always point into *to-space*. Each time a value is loaded into a tended descriptor by reading from an internal field of a dynamically allocated object, the value is tended before being assigned to the tended descriptor.

The garbage collection system imposes no restrictions on the sizes or internal organization of dynamically allocated objects. It requires, however, that the application make known to the garbage collector which words within allocated objects represent pointers to other objects. Since the live objects copied into *to-space* may themselves contain pointers to other live objects, it is necessary to tend all of the descriptors within these objects as they are being copied. Typically, objects are first copied, and then scanned.[2] The scanning process examines the pointers contained within each copied object and makes sure that all of the objects referenced by these pointers are also copied into *to-space*. After arranging for the referenced objects to be copied into *to-space*, if they have not already been scheduled for copying, the scanner updates the pointers to these objects to reflect their new locations.

In order to support fast response to memory read, write, and allocate instructions, it is necessary to divide the garbage collection process into a number of very small atomic actions. Certain system invariants are maintained between execution of these atomic actions. These invariants are sufficient to allow memory read and write operations to interleave with background garbage collection efforts. Because there is no limit on the size of objects supported by the garbage collector, it is essential that copying and scanning of objects be performed incrementally. Otherwise, the time required to complete a single atomic operation might exceed the desired real-time response. When an object is scheduled for copying, memory is set aside for the copy in *to-space* and pointer links are established between the new and old locations of the object. The *to-space* memory region is divided into several subregions by pointers that represent the boundaries between different memory subregions. Hardware assisted range checks are performed on the addresses that accompany each memory operation to determine which subregion is being accessed. The organization of *to-space* is illustrated in Figure 2.1.

Special handling is required each time an attempt is made to read memory found between the scanned and reserved pointers. Whenever the application attempts to read memory that has been copied but not yet scanned, the garbage collector must

[2]The architecture simulator used for the experiments reported in this dissertation actually scans most objects while they are being copied. Scanning of slice data regions (see reference [40]) is delayed until copying is completed.

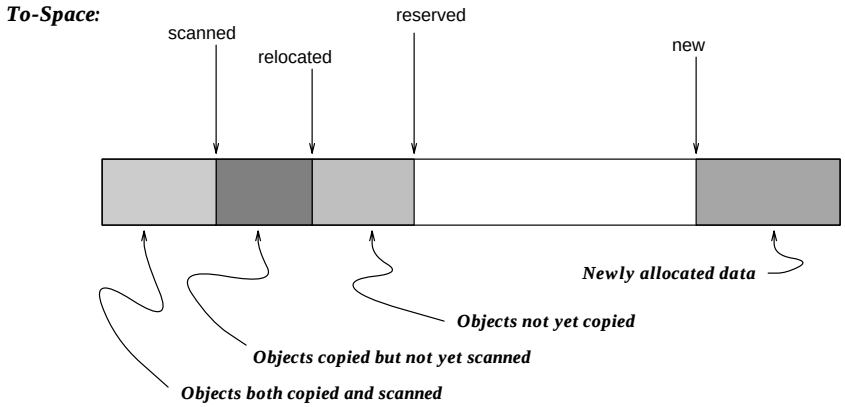

Figure 2.1: Organization of *to-space*

scan the requested data before making it available to the reading process. Scanning consists of first determining whether the requested data represents a pointer or raw data. Then, if the data is a pointer, the garbage collector tends the pointer before returning its value. Whenever the application attempts to read data that has not yet been copied, the garbage collector must determine the location of the source object residing in *from-space*, and return the data from the appropriate location in *from-space* after first scanning it. Certain write operations also require special handling. Whenever the application writes to memory reserved for copying, but not yet copied, the garbage collector redirects the write operation to the appropriate address within the uncopied object still residing in *from-space*.

As mentioned above, the Ellis, Li, Appel garbage collector [10] requires approximately 100 msec to perform a flip, and 50 msec to read an unscanned object. This means that a hard real-time scheduler must assume that every object allocation and every read will require at least 100 msec or 50 msec, respectively, to complete. This is clearly unacceptable for a large class of time-critical applications. The special hardware described in the following section offers a worst-case memory access time of six traditional memory cycles. The time required for a flip ranges between 5 and 50 μsec, depending on certain configuration-specific parameters, such as the speed of memory and the number of descriptors that the mutator has to tend.

2.2 Object types

The garbage collection architecture supports three types of objects: *records*, *slices*, and *stacks*. Slices and stacks are further subdivided into *descriptor* and *terminal* varieties, distinguished by whether or not they may contain pointers into garbage-collected memory. Each object is preceded by a small header, the size and format of which depends on the object type. For records and slices, the header consists of one word identifying the type and size of the object. Headers for stack objects contain more information, as detailed below. For every object, an additional bit of memory per word identifies which words in the object currently contain descriptors. Note that these tag bits are dynamic in the sense that a word may contain terminal data and a descriptor at different times during program execution.

A *record* is a fixed block of storage containing any combination of descriptors and terminal data. The record is the basic object type used to implement practically all language data types, including structs and unions in languages such as C; Smalltalk and C++ objects; and Lisp dotted pairs. Complex data structures may be built by linking together objects constructed from records.

In addition to its header, a *slice object* contains only two fields: a length field and a pointer to a location within a region of *slice data*. The length field indicates how many contiguous bytes of slice region data are contained within the slice. Slices are useful in implementing the built-in string and stream data type of languages such as Icon [14] and Conicon [37, 38]. They are also useful in any context where a "fragmentable array" data abstraction is pertinent, for instance in editing of audio or video data. Once allocated, a slice object is considered to be read-only. Only the slice region data referenced by the slice object is writable.

Once a slice object has been allocated, *subslices* of the object may be allocated as well. A subslice is merely a slice object that points to slice region data originally allocated for another slice object. For example, a slice object might refer to slice region data containing the string, "Hello, world". A subslice of this object, beginning at position 3 (using zero-based indexing) and having length 2, would refer only to the string "lo". Slice data regions have the unique property that they can be *partially collected*; that is, if any portion of slice region data is no longer referenced by any

slice object, the garbage collector reclaims that portion while the rest of the slice region continues to exist. In the above example, if the "Hello, world" slice object were no longer reachable from a chain beginning at any tended descriptor, it would be reclaimed by the collector. Additionally, the slice region portions containing "Hel" and ", world" would no longer be reachable either, so they would also be reclaimed. All that would remain would be a single slice object and two bytes of slice region data.[3]

A *stack* is a fixed-size object containing descriptors and terminal data, together with an extra header word indicating where the current top-of-stack is located. In the current simulated prototype, all stack objects grow downward. Stack objects are useful for implementing run-time activation stacks, providing a simpler mechanism for function calls than the alternative of heap-allocated activation frames. Stack objects are discussed in more detail in section 4.1.4.1; tradeoffs between stack objects and heap-allocated activation frames are investigated in chapter 6.

2.3 The garbage-collected memory module architecture

2.3.1 Overall system architecture

The proposed memory system is designed to make effective use of state-of-the-art central processing units and standard architectures, as shown in Figure 2.2. The garbage-collected memory is a shared resource of all real-time processes and is accessed using physical (rather than virtual) memory addresses; processes that do not have stringent timing requirements may use standard storage management techniques in less expensive RAM. The use of physical memory addressing is consistent with common practice for time-critical processes, which must remain memory-resident throughout their lifetimes and which require a rapid context-switching mechanism. For simplicity, this section describes a uniprocessor configuration, although minor modifications to the design would easily support the use of bus-based, shared-memory multiprocessors.

[3]This example should not be taken too literally. In practice, the granularity of live data is along word boundaries, so that a few extra bytes that would otherwise be garbage may be retained at each end of a subslice.

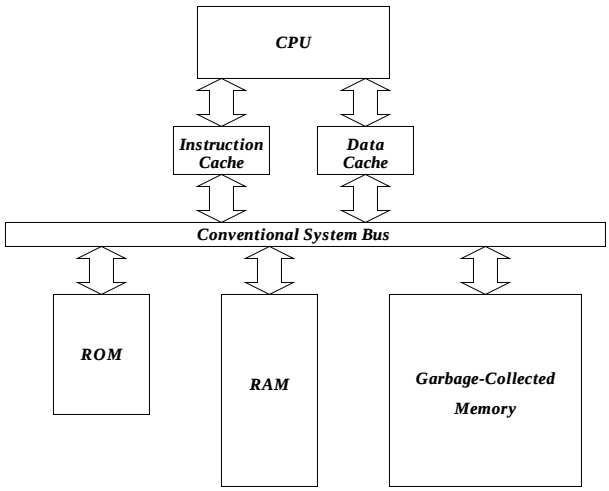

Figure 2.2: System architecture

The garbage-collected memory module presents the illusion of being normal memory. The highest-priority task of the module is to field and service memory stores and fetches issued by the CPU. The module also responds to several I/O addresses used by the CPU to issue commands and receive responses; this is discussed in more detail in section 4.1.1.

Figure 2.3 illustrates the garbage-collected memory module in more detail. The module is based around a central internal bus, connected to the system bus via a *bus interface unit* (BIU). The module also includes a dedicated microprocessor that continuously performs the algorithm of section 2.1, leaving the central processor free for normal processing. Memory references by both the central processor and the garbage-collection processor travel along the internal bus to the memory banks, depicted in the figure as RAM1 and RAM2; at any time, one of these banks contains *to-space* and the other contains *from-space*. The *memory arbiter* (a preliminary version of which has been described in reference [45]) snoops on the internal bus and intercepts requests that require additional processing because of ongoing garbage collection activity. If necessary, the BIU provides handshaking signals to stall the CPU until intercepted requests have completed.

Most memory requests issued by the CPU to the garbage-collection module are

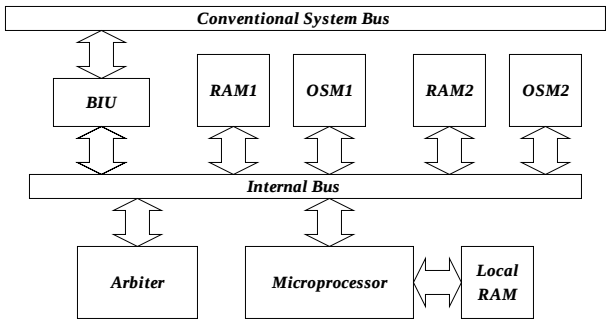

Figure 2.3: Garbage-collected memory module internal architecture

handled in the same time as traditional memory. During active garbage collection, however, delays may be imposed by contention between garbage-collection activities and the CPU's request. Additionally, memory operations that reference locations not yet scanned or locations within objects that have been queued for copying but not yet copied require additional memory cycles. Rather than interrupt the CPU to handle these requests, the CPU is stalled using traditional bus wait states. The maximum delay for a particular memory operation is approximately six traditional memory cycles (see references [40, 41] for details).

Within the garbage-collection module, three distinct threads of control run concurrently. Two of the threads run on the arbiter, and the third is executed by the garbage-collection microprocessor. The division of labor between the arbiter and the garbage-collection microprocessor represents tradeoffs between cost and performance. To provide fast response to CPU requests, all CPU services and many background services must be implemented entirely by the arbiter. To reduce costs, all services that do not need to be hardwired into the arbiter are handled by the garbage-collection microprocessor, which is a stock component. Hardwiring of arbiter services permits the worst-case time required to interrupt all background garbage-collection activities to be kept within approximately one memory cycle.

2.3.2 Motivation for the object-space manager

The object space manager described in chapter 3 and reference [39] is needed to find the header information associated with objects. This header must be found and read in the following situations:

1. Each dynamically allocated object must make its internal organization available to the garbage collector so that raw data bits can be distinguished from pointers to other objects. This is done either by tagging each word of the object independently, or by encoding the object's organization in its header. If the latter alternative is used, then header lookups are required each time an attempt is made to read from unscanned objects residing in *to-space*. The implementation investigated here explicitly tags each word in an object.

2. If an attempt is made to read from or write to an uncopied object, the header of the uncopied object identifies the true location of the object in *from-space*.

3. Descriptors do not necessarily point at the headers of the objects they refer to. Instead, they frequently point at internal fields within these objects. Each time a descriptor pointing into *from-space* is tended, the header of the referenced object is consulted to determine the total size of the object and to decide whether the object has already been scheduled for copying. If the object has been scheduled for copying, the header points to the object's new location in *to-space*.

Since the size of an object is theoretically unlimited, it is not feasible for software to provide constant-time access to the data base of header locations. Either the time required to locate a header, or the time to install a new header into the data base, is proportional to the size of the objects involved. Stapleton [48] solved the header lookup problem by using small software "crossing maps," precursors of the OHB and CAR registers described in section 3.3. Stapleton's crossing map consists of two words for every 32 words of garbage-collected memory. One of these words contains one bit for each of the 32 words; a set bit indicates that a data object begins at that address. The other word contains a pointer to the beginning of the data object (if any) that crosses the starting boundary of the 32-word segment. In this case, locating

a header is a constant-time operation, but the proportional penalty must still be paid when the crossing maps are updated to reflect a new header. This occurs not only upon the initial allocation of an object, but also whenever it is copied into *to-space* following a flip.

Crossing maps were also used in the page-fault technique developed by Ellis, Li, and Appel [10]. Here the crossing map consists of a bitmap where each bit corresponds to a page of virtual memory in the heap address space. A bit is set if an object spans the beginning of the corresponding virtual memory page. To find the header of an object that spans a page boundary, their algorithm scans back page by page until a page is found that begins with a fresh object. This is the desired object, since only objects larger than a page are permitted to span page boundaries. In this scheme, the proportional penalty is paid both with each object creation (to update the crossing map) and with each header lookup (to scan the crossing map); however, this penalty is only significant for very large objects. A more important issue in their scheme is the high latency for references to unscanned data.

2.3.3 Alternatives to the object space manager

One prevalent school of thought is that the header location problem should be circumvented by preventing arbitrary access to internal addresses of objects; all accesses to heap objects should consist of the base address of the object together with an offset into the object. This view, however, ignores the fundamental reason that real-time garbage collection methods are out of favor with system implementors: methods provided to date have simply been too inefficient, as discussed above. To require this base-offset form of addressing is to preclude a number of gainful optimization techniques, such as strength reduction and induction variable elimination, thus exacerbating the efficiency problem. Such optimizations are very important for applications using general-purpose garbage collection, in which string and array objects can be allocated from the heap. Base-offset addressing doubles the number of registers and memory cells required to represent pointers, and doubles the number of memory cycles required to fetch or store derived pointers. Additionally, the underlying memory architecture would either have to support the base-offset paradigm in hardware by widening the system bus to contain wires for both a base address and

an offset, or else memory writes would require two memory cycles each in order to send both the offset and the data to the garbage-collected memory module. Memory reads would also require two cycles each unless the hardware protocol provided for placing the offset on the data channel during the read request. Thus either reads and writes would become very expensive, or the arbiter protocol would become much more dependent on the machine organization than is necessary in the design presented here, which assumes the use of stock workstation components. Machine dependency is economically undesirable, since the development costs of vendor-specific hardware cannot be shared as widely.

One variation on base-offset addressing is a technique sometimes called "Red/Pink register pairs" [57], used originally in PDP10 MacLisp. In this approach, derived pointers are supported by using a "Red" register to store the base of the object and a "Pink" register to hold the derived pointer. During garbage collection, a runtime check is made to determine if the Pink register currently contains a derived pointer into the object referenced by the Red register. If so, the collector relocates the object and updates both registers. Costs of this technique include wasted registers, compiler complexity, and the run-time cost of loading the Red registers. Special handling is required for derived pointers that must exist outside of registers, for instance when such pointers are to be passed as parameters on the call stack. The collector must also be informed which registers are organized as Red/Pink pairs and treat them differently from other registers. Alternatively, normal and derived pointers can be treated identically by storing *all* pointers as a base and an offset, but this is wasteful of memory and further reduces the available register pool.

If efficient optimizations are to be permitted using standard pointers, some means must be provided for rapidly locating the header of an object given a pointer to any location within the object. Of course, it would be unacceptable for each memory allocation or heap reference in a real-time program to incur a penalty that is potentially proportional to the size of the largest object in the heap. As discussed above, however, existing software solutions must pay such a penalty either at each reference or whenever an object is copied into *to-space*. This suggests that achieving both constant-time performance and efficiency requires hardware assistance. The memory architecture model presented here is designed to satisfy these conflicting goals.

3. THE OBJECT SPACE MANAGER

3.1 Design criteria

A primary component of the proposed architecture is the OSM chip, which provides an abstract view of garbage-collected memory as an *object space*, i.e., a collection of non-overlapping *objects*. An object here is simply a contiguous block of memory having a starting address and a length; the contents of an object are of no importance under this abstraction. The OSM provides a mapping mechanism between raw addresses and objects. This section concentrates on the description and analysis of the OSM design; other components of the garbage collection architecture are described in [40, 41, 45] and in chapter 2 of this dissertation.

The following goals were set forth for design of the OSM:

- The chip should support a small, but general, set of commands for object manipulation. Since the object space abstraction may be useful in other applications, it is important that this command set provide *mechanisms* for object manipulation, but not *policy* for the use of the object space. No assumptions should be made about the nature of the hardware (hereafter called the *client*) that makes use of the object space.

- The chip should be as system-independent as possible. Thus interfacing to the chip should be by a well-defined protocol that is convenient and easily implemented.

- After functionality, speed is of primary importance. All commands recognized by the chip should be implemented using the least possible propagation delay. Since object lookup is expected to occur far more frequently than object cre-

ation or deletion, chip operations should be optimized in favor of lookups, if such choices become necessary.

- The cost of the object view of memory should be as low as possible. The design should therefore minimize the amount of circuitry required per word of the object space. Naturally, there are tradeoffs between speed and circuit size; therefore, different alternatives emphasizing each should be explored.

- The object space should be scalable. The design should allow a large object space to be controlled by a number of cooperating replicas of the chip, each controlling a subspace of the whole.

To summarize, one would like a fast, inexpensive chip, well suited for expansion and having broad functionality. Of course, all these criteria are in conflict with one another. The design presented here is the result of exploring various alternatives and discarding those that do not form an acceptable compromise among the various goals. (For a discussion of alternatives and why they were rejected, see section 3.6.)

3.2 Interconnection architecture and command interface

Recall that the garbage-collected memory is composed of two separate banks. At any point in time, one of these banks contains *from-space* and the other *to-space*. An object view of each bank is provided by an object space manager, whose purpose is to keep a record of where valid objects are currently located in its assigned memory bank. Since the garbage-collected memory is shared among all real-time processes, each bank will generally be larger than can be controlled by a single OSM chip; thus a number of OSM chips generally cooperate, using a single local bus that connects them to each other and to the memory arbiter (see Figure 3.1). It is convenient to refer to all cooperating chips on a bus as a single OSM, regardless of their number. The address space controlled by one chip is referred to as its *chip space*.

Command set. The OSM was originally envisioned as having an extensive command set, with sophisticated error checking capabilities. As discussed in section 3.6, it turns out that providing more than a few basic features is quite expensive and slows the performance of the chip. Fortunately, only a few commands are needed to provide a

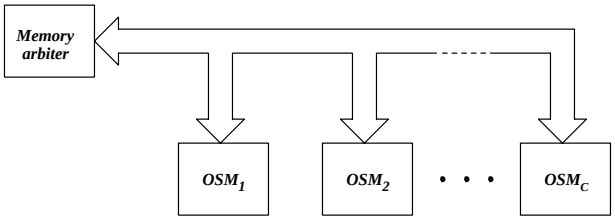

Figure 3.1: OSM interconnection

number of powerful services. The philosophy of the design presented here is that the client can be trusted not to abuse the features provided; therefore, error checking is unnecessary.

The following functions are performed by the object space manager.

1. Create object. The client provides the beginning and ending addresses of the space to be occupied by the object. The OSM then maintains this information until the object is deleted.

2. Delete object. The client provides the beginning address of the object to be deleted. If an object is indeed located at the indicated address, the OSM notes that it no longer exists.

3. Return header address. The client provides an address, and the OSM provides the address of the header of the containing object. If the address is not contained inside any known object, the result is undefined.

4. Clear chip space. This command deletes all record of objects controlled by one or more chips. This command should be used with caution when multiple chips are used to control a large space. It is the client's responsibility to ensure that use of this command does not cause an object that spans more than one chip to be only partially deleted.

This command set, although limited, is more than sufficient for the purpose of real-time garbage collection. The Create object command is used to allocate space for new objects, and when copying objects from *from-space* into *to-space*. The Return

header address command permits rapid access to the bookkeeping information stored in object headers, allowing quick tracing of pointers between *from-space* and *to-space*. The Clear chip space command is used to delete all objects in *from-space* at the time of a flip, preparing the associated memory region to become *to-space*.

The Delete object command was originally included because it was believed at the time that it would permit certain garbage-collection activities to be performed more efficiently. For example, it was thought that better storage utilization would be achieved by merging certain kinds of small objects into larger objects as part of the garbage collection process. During garbage collection, contiguous space would be reserved for the independent copying of each of the small objects. In order to handle memory operations that referred to these objects while they were being copied, each object would be given a header that pointed to the source from which it was being copied. Later, after all of the small objects had been copied, they would be merged into a single object with a shared header by sending Delete object commands to the OSM for each of the small copied objects, and sending a Create object command to the OSM for the resulting merged object. This no longer seems worthwhile because of certain technical concerns, which are discussed in more detail in section 3.6, below.

Interfacing. The pinouts for a single OSM chip are shown in Figure 3.2. The signals are divided into two groups.

1. Client-OSM interface. The three COM lines are used by the client to encode the command to be executed. The d DATA lines are used to pass the addresses required as parameters of each command, as well as to return the header address of an object to the client. The $\overline{\text{ACK}}$ line is used to return positive acknowledgments; the reason for its inversion is explained below. Finally, the CLOCK signal is used to implement the clocked asynchronous client-OSM protocol and to provide timing for the internal operations of the OSM.

2. External signals. The RESET signal is used to place the OSM in its initial state on power-up. The CHPSEL signal is not used under normal operation; rather, it is used to enable modification of a chip's internal c-bit CHPADR register via some of the DATA lines. The CHPADR register specifies the c high-order bits of each address controlled by an individual chip.

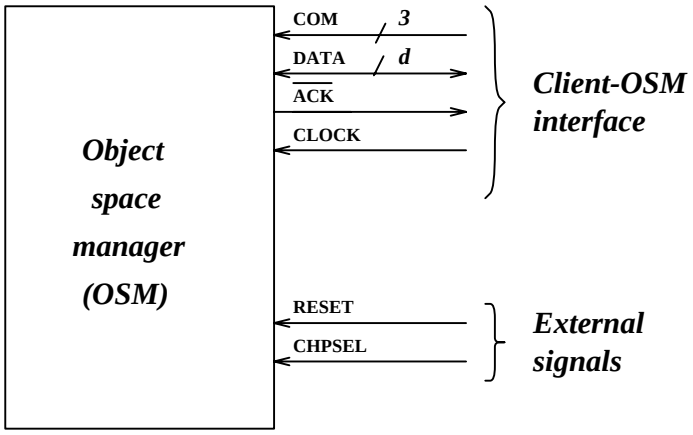

Figure 3.2: OSM pinouts

Since several chips on the same local bus may share responsibility for an object space, the question of which chip should acknowledge a command is problematic. This is especially true since a single command may be serviced by more than one chip. For instance, consider the creation of an object that spans the boundary between two chip spaces. After the object is created, at least one of the chips must lower its $\overline{\text{ACK}}$ line to inform the client that the service has completed.

Arbitration of acknowledgments is handled using wired logic [5]. All of the chips along one bus tie their $\overline{\text{ACK}}$ pins to a single common line, using open-collector (no-pullup) drivers. A common line driven by no-pullup drivers has the effect of ANDing the various signals placed on it; if both the inputs and the output are inverted, this logic is changed to an OR. Thus the problem of which chip acknowledges is solved by having all involved chips acknowledge every command. Each chip normally broadcasts logic 1 on the $\overline{\text{ACK}}$ line. Whenever any chip has completed the service, it lowers its output to so indicate. The timing program in each of the chip control units ensures that all affected chips will respond during the same clock cycle.[1]

Command protocol. Communication between the client and OSM employs a clocked asynchronous protocol. For all services, the client places a command on the COM lines and some information on the DATA lines. The client then holds this

[1] If the wired-AND logic is infeasible in certain configurations because of fan-in or fan-out constraints, it can be replaced by SSI logic without change to the chip design.

information steady on the lines until it receives an ACK response. For some of the commands, two different words must be transmitted to the OSM; in this case, the first DATA word is held steady for one clock cycle. Then, the second DATA word is held on the lines until the requisite time has passed. After receiving an acknowledgment, the client must drop the COM lines to logic 0 for at least one clock cycle before issuing a new command. This indicates to the OSM chips that the acknowledgment was received, and allows them to enter their idle state. For the Return header address command, the address is returned on the DATA lines at the same time that the $\overline{\text{ACK}}$ line is driven low.

3.3 Some design details

The following discussion makes frequent reference to a few basic parameters. Let the number of data blocks[2] in the object space be $D = 2^d$; thus d bits are required to specify the address of any object. Also let the number of OSM chips that share control of the object space be represented by $C = 2^c$, and the number of data blocks in an individual chip space by $W = 2^w$. Obviously $w = d - c$.

The OSM chip circuitry can be logically divided into four parts:

(1) a *control unit*, which implements the command protocol and is responsible for other timing considerations;

(2) a pair of *data registers*, which store information about the whereabouts of objects in the object space;

(3) the *access tree*, which communicates the commands to the appropriate portions of the data registers and reports information back to the control unit; and

(4) the *tree-register interface*, which passes information between the access tree and the data registers.

[2]The amount of memory controlled by a single OSM chip can be adjusted by varying the granularity of the object space. This is discussed at greater length in section 3.4. In the current discussion and analysis, each word is treated as a distinct block.

Figure 3.3 shows the relationship between these logical divisions. Note that although the tree-register interface and OHB are shown as conceptually separate components, the OHB register is physically contained within the tree-register interface. Also, the logical structure depicted in Figure 3.3 should not be confused with a physical layout. The access tree would actually be implemented using a "hyper-H" layout such as the one presented by Leiserson [27], in which each bit of the OHB (and the tree-register interface) would be located at some distance from its "neighbors."

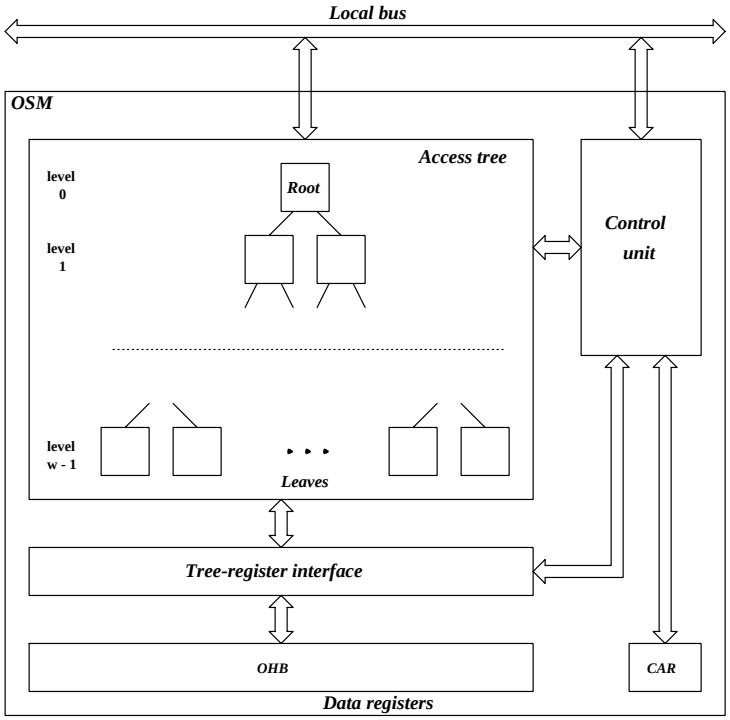

Figure 3.3: OSM structure

Control unit. Figure 3.4 shows the interface details of the control unit. The line labeled CREATEOBJ indicates to the tree-register interface that a new object is to be created. The CLEAROHB and CLEARCAR lines are used to clear all data from the OHB and CAR registers (described below), respectively. This occurs for both registers at chip reset and whenever the client requests the Clear chip space service; the CLEARCAR line alone is asserted when the object whose header appears in the CAR

is deleted. The STROBEOHB and STROBECAR lines allow clock pulses through to the registers only when these registers are to be updated. The DATENABLE signal controls write access to the DATA lines of the local bus. When DATENABLE is asserted, the OSM writes to the data bus; otherwise, it reads from the bus.

The construction of the control unit is straightforward and is not discussed in this dissertation.

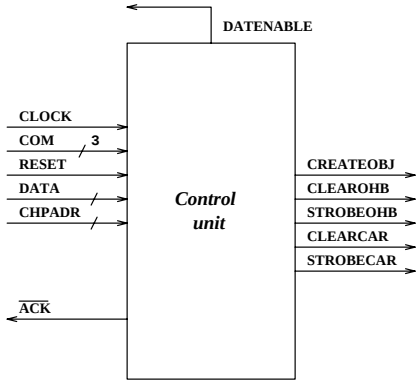

Figure 3.4: Control unit

Data registers. The object space manager maintains information about where objects are located in the object space by using several registers. The W-bit *object header bitmap (OHB)* contains one bit for each block controlled by the chip; if this bit is set, it indicates that an object header begins at that block. All objects must be block-aligned. Another register is used to handle objects that cross chip-space boundaries. The *crossing address register (CAR)* contains the d-bit absolute address of the beginning of the object that contains the first block of the chip space, if such an object exists and begins on another chip.

There are two additional registers not shown in Figure 3.3. For those commands requiring transmission of two addresses over the DATA lines, the *hold register (HR)* buffers the first address transmitted. The HR is located in the control unit. The CHPADR register is discussed in section 3.2, above.

Access tree. In order to provide rapid access to the huge OHB register using a minimum amount of circuitry, a binary[3] tree propagates commands from the control unit to the registers and returns responses to the control unit. The root of the binary tree communicates with the control unit and with the client. The leaves are connected to the OHB via the tree-register interface. Each leaf is conceptually "responsible" for two blocks of the address space. Each internal node is responsible for that portion of the address space for which any of its descendants are responsible. Thus the access tree is a device for dividing up this responsibility, providing rapid, heavily parallel access to the OHB.

Level 0 of the access tree consists of the root, and level $w - 1$ contains the leaves. All nodes at a particular level are exactly alike, but each level of nodes differs slightly from the previous one. This is because address decoding is performed within the access tree. The root node examines the most significant bit of each address to determine in which half of the address space it resides, and passes the remainder of the address to the appropriate child node. The child then examines the most-significant remaining bit, and so on. Each leaf has only one bit of address remaining, indicating which of its two controlled blocks (if either) has the correct address.

The OSM must also be able to generate the address of the header of an object. In this case, each leaf generates the least-significant bit. The next level selects a leaf and adds another bit, and so on. The difference in node circuitry at various levels of the tree is entirely due to the number of address bits that have been encoded and remain to be decoded at each level.

Tree-register interface. It is helpful for all nodes of the access tree to be uniform, with the exception of the addressing differences just described. At the leaf level, however, the final signals must be used to update the OHB or return address bits back up the tree. The circuitry to perform these tasks is referred to as the tree-register interface.

[3]Of course, the fanout of the tree need not be binary. Section 3.5 discusses the pros and cons of different fanout degrees.

3.3.1 Command set implementation

Figure 3.5 shows the internal logic of a single node at level k in the access tree, and Figure 3.6 shows the tree-register interface circuitry for a typical bit of the OHB register (hereafter called a *slice*). Note that each access tree node is divided into two sections. The logic pictured above the dashed line in Figure 3.5 is identical no matter where the node occurs in the tree, while the logic beneath the line is replicated for each j such that $k < j < w$. The STROBEOHB signal in Figure 3.6 is a masked clock signal. The control unit allows the clock pulse through only when an object is to be created or deleted. The remainder of this section describes how the illustrated circuitry supports each service in the command set.

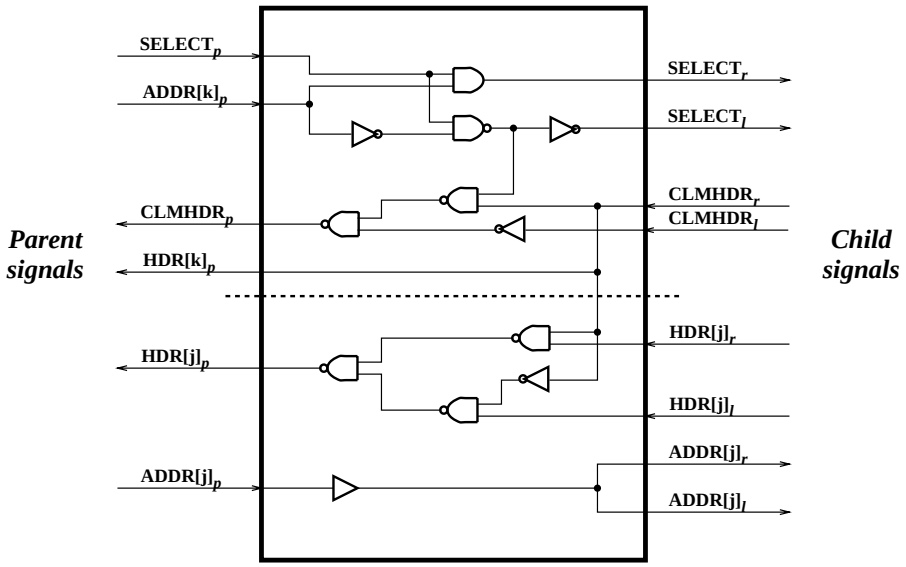

Figure 3.5: Logic diagram for access tree node at level k

Create object. When a client requests the Create object service, each chip's control unit determines if either or both of the delimiting addresses of the new object are located within its chip space. If the beginning address is local to a chip, the control unit asserts the SELECT$_p$ line of the root node of the access tree. If the chip space does *not* contain the beginning address, but all or part of the chip space is contained in the new object, the address is stored in the CAR.

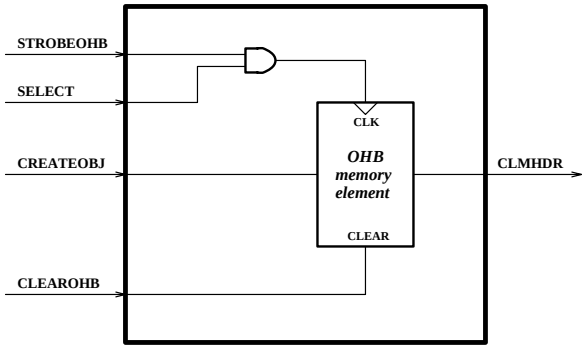

Figure 3.6: Typical slice of the OHB register interface

Provided the beginning address of the object is controlled by the chip, the SELECT logic in Figure 3.5 decodes one bit of the starting address at each level of the tree to locate the matching bit of the OHB. If a level-k node's parent signal, SELECT$_p$, is asserted, it passes this signal to exactly one of its children via either the SELECT$_r$ or the SELECT$_l$ line. The child selected depends on the value of the k^{th} bit of the address.

At the frontier of the tree, exactly one element of the tree-register interface receives an asserted SELECT signal (see Figure 3.6). The CREATEOBJ signal indicates to all slices of the interface that a new object is to be created. The selected slice updates its OHB cell contents to logic 1, and all other elements retain their previous values.

Delete object. The object deletion service uses the same SELECT logic as Create object to locate the affected bit of the OHB. In this case, the CREATEOBJ signal is not asserted, so the selected slice updates its OHB cell contents to logic 0. All other elements retain their previous values.

Return header address. This service also shares the SELECT logic with the previously discussed commands, using it to locate the address supplied by the user. The chip must then find the first bit to the left of the selected bit of the OHB that contains a value of 1, and construct the address associated with that bit. The method to do this is quite simple: each element of the OHB whose value is one presumes it is the bit that is sought. It indicates this to its leaf of the access tree using the CLMHDR

signal, as shown in Figure 3.6. Each node of the access tree then determines whether it could possibly contain the header, by examining the CLMHDR signals of its children and the SELECT signals computed previously. If so, it in turn asserts its own CLMHDR signal to its parent.

The logic used to determine the CLMHDR signal at a node is

$$\mathrm{CLMHDR}_p = \mathrm{CLMHDR}_l \vee (\mathrm{CLMHDR}_r \wedge \overline{\mathrm{SELECT}_l}).$$

Since

$$\overline{\mathrm{CLMHDR}_p} = \overline{\mathrm{CLMHDR}_l} \wedge (\overline{\mathrm{CLMHDR}_r} \vee \mathrm{SELECT}_l),$$

this means a node claims to have the header unless (a) neither of its children does, or (b) its left child contains the address supplied by the client but does not claim to have the header. In the latter case, the header obviously cannot be within the right child.

While this computation is taking place, each node is also building up the address where the header is located. A node at level k of the access tree constructs the least-significant $w - k$ bits of the address as follows. Whenever a node claims to have the header, it determines which of its children it believes contains the header, and passes the $w - k - 1$ least-significant bits of the address from that child to its own parent. (These signals are illustrated as HDR[j] in Figure 3.5.) The $(w-k)^{\mathrm{th}}$-least-significant bit (HDR[k]) is set to zero if the left child is selected, and one if the right child is selected. If a node does not claim to have the header, the returned address signals are treated as "don't-cares." The header address returned to the client consists of the contents of the c-bit CHPADR register concatenated with the w HDR bits produced at the root of the access tree.

It may be that the header of the object with the selected address does not reside in the same chip space as that address. When this occurs, the CLMHDR signal produced at the root of the tree will be zero, since all bits of the OHB to the left of the selected address must be zeroes. Instead of placing the HDR signals on the DATA lines, the OSM returns the contents of the CAR register, which were gated in when the object was created. The CAR register allows the header of an object to be returned within a fixed interval, regardless of the number of chip spaces spanned by the object.

Clear chip space. The access tree is not involved in performing this command. The control unit asserts the CLEARCAR and CLEAROHB signals, which immediately reset the contents of the CAR and OHB registers to zero.

3.4 Analysis

3.4.1 VLSI technologies

Advances in VLSI technology have led to a bewildering variety of available design styles. (For an introduction to issues of VLSI design, see for example [12, 32, 56].) Any analysis of chip areas and propagation delays must assume particular technology choices. NMOS is capable of high chip densities, but is subject to power consumption that is prohibitive for an application such as the OSM. Traditional CMOS achieves very low power consumption, but requires nearly twice as much chip area for the same functionality as does NMOS. However, there are a number of CMOS variations that combine low power, high speed, and low transistor counts at the expense of design complexity. Two of these are dynamic CMOS and domino CMOS [4, 23].

A typical n-input NMOS gate requires $n+1$ transistors to implement, while the same CMOS gate requires $2n$ transistors. Dynamic and domino CMOS techniques [4, 23] are designed to achieve the low power consumption of CMOS and the low transistor counts of NMOS by using precharging and clocking techniques. However, these methods only approach the transistor counts of NMOS in the limit: n-input gates each require about $n+3$ or $n+4$ transistors, so dynamic and domino CMOS only provide area savings for fairly large gates. Furthermore, dynamic and domino CMOS circuits are more difficult to design. Dynamic CMOS techniques require multiphase clocking for cascaded circuits; domino CMOS can use a single clock edge, but is not capable of expressing negations.

The purpose of the next section is to estimate the number of transistors required to implement the OSM. While NMOS is impractical for such an implementation, it is useful to estimate transistor counts for NMOS as a lower bound on what can be achieved using dynamic and domino CMOS techniques. Similarly, traditional CMOS provides an upper bound on transistor counts. As the analysis shows, the number of blocks W controllable by a single chip is identical for either technology, since W

must be a large power of 2. Thus the OSM may be implemented using whatever combination of CMOS design styles provides the best speed advantage.

3.4.2 Transistor counts

The binary access tree of the OSM provides a highly regular structure, simplifying layout of the circuitry. A binary tree can be implemented in an area-efficient manner, particularly when the number of connections between nodes decreases as one approaches the leaves [27, 28]. The use of a hyper-H layout such as that described by Leiserson [27] also minimizes wire area and wire delays.

This section analyzes the size of an object space that can be controlled by a single OSM chip. As a point of reference, assume that an OSM chip has the same die size as a one-megabit DRAM chip, and assume further that 0.5μ technology (roughly equivalent to that used in the DRAM chip) is available for the OSM chip implementation. The DRAM chip can be implemented using just over $3\text{M} = 3 \cdot 2^{20}$ transistors (see Taub and Schilling [51] for details), plus additional routing area. Although the DRAM and the OSM both benefit from very regular layouts, the hyper-H tree layout of the OSM cannot use area quite as efficiently as the grid arrangement of a DRAM. Even if an additional one-third of the die area were completely wasted, however, at least 2M transistors would be available for implementation (as a conservative estimate).[4]

As mentioned above, different technologies permit different numbers of transistors per gate. Table 3.1 contains a list of parameters used in this discussion, each of which is the size of a circuit element in transistors.

Recall that $W = 2^w$ represents the number of blocks of memory controlled by the chip. Clearly

[4] Of course, the transistor densities achievable for a custom chip are not as great as those of DRAM chips, even when the same device size is used. This is due to special custom processes that are only cost-effective for chips produced in large volume. The OSM chip must be produced with a smaller device size to equal the density of a DRAM. The present analysis assumes availability of 0.5μ technology, which will soon be standard [42].

Table 3.1: Transistor cost parameters

Parameter	**Circuit element**
N_{invert}	inverter
N_{gate}	two-input gate
N_{FF}	D flip-flop with clear
$N_{\text{node}}(k)$	node of the access tree at level k
N_{slice}	slice of the tree-register interface
N_{buff}	buffering of signals to tree-register interface
N_{tree}	entire access tree
N_{interf}	entire tree-register interface, including the OHB
N_{control}	control unit
N_{chip}	entire chip

$$\begin{aligned} N_{\text{chip}} &= N_{\text{control}} + N_{\text{tree}} + N_{\text{interf}} \\ &< W + \left[\sum_{k=0}^{w-1} 2^k N_{\text{node}}(k) + N_{\text{buff}} \right] + W \cdot N_{\text{slice}}. \end{aligned}$$

Here the control circuitry is estimated to be less than W transistors, since W is a large number and the control circuitry is negligible; this is merely to simplify the analysis. To estimate $N_{\text{node}}(k)$, recall that there are $w-k-1$ copies of the circuitry beneath the dashed line in Figure 3.5. Furthermore, when $w-k-1$ is large, some buffering of the CLMHDR$_r$ signal and its complement is necessary. If we impose a maximum fanout degree of eight to reduce propagation delay, the CLMHDR$_r$ signal must be buffered once for each additional seven ADDR lines after the first seven, at a cost of $2N_{\text{invert}}$ transistors per buffer gate. Its complement may be tapped off from between the two inverters that constitute each of these buffers. Each node at level k thus requires $\lceil (2/7)[(w-k-1)-7] \rceil N_{\text{invert}} < (2/7)(w-k-1)N_{\text{invert}}$ transistors to buffer CLMHDR signals.

The number of transistors contained in a node at level k can now be estimated by counting gates in Figure 3.5. (The ADDR buffer gates are handled separately below.) By inspection and the above discussion,

$$N_{\text{node}}(k) < a + b(w-k-1),$$

where $a = 4N_{\text{gate}} + 4N_{\text{invert}}$ and $b = 3N_{\text{gate}} + (9/7)N_{\text{invert}}$. (Note that the AND gate in Figure 3.5 counts as a gate and an inverter.)

The parameter N_{buff} of Table 3.1 refers to the buffer gates on the ADDR lines in Figure 3.5, as well as buffering for the STROBEOHB, CREATEOBJ, and CLEAROHB signals needed at the frontier of the tree. Again assuming a maximum fanout degree of eight, the address and other lines need not be buffered more frequently than every third level of the tree. Obviously the greatest benefit is achieved by not placing buffer gates on the bottom two levels of the tree, where about 3/4 of the nodes in the tree are located. Therefore this analysis assumes that buffers are placed in all nodes at every third level, beginning at the third level from the bottom.

Renumber the levels of the tree such that the third level from the bottom is level 0, the fourth level from the bottom is level 1, and the root is level $w-3$. At the i^{th} level, there are buffer gates only if $i \bmod 3 = 0$. There are 2^{w-3-i} nodes at level i. The number of address lines passing through level i is $i+2$; there are an additional three lines; and each buffer gate requires $2N_{\text{invert}}$ transistors. Thus the total number of transistors required for buffering is

$$\begin{aligned}
N_{\text{buff}} &= \sum_{i=0}^{w-3} 2N_{\text{invert}}(i+5)2^{w-3-i}[\![i \bmod 3 = 0]\!] \\
&= \tfrac{1}{4}N_{\text{invert}} \sum_{l=0}^{\lfloor\frac{w-3}{3}\rfloor} (3l+5)2^{w-3l} \\
&= \tfrac{1}{2}W \left[\sum_{l=0}^{\lfloor\frac{w-3}{3}\rfloor} (3l)2^{-3l} + 5 \sum_{l=0}^{\lfloor\frac{w-3}{3}\rfloor} 2^{-3l} \right] \\
&< \tfrac{1}{2}W \left[\tfrac{5}{8} + \tfrac{185}{32}\right] \\
&< \tfrac{7}{2}W,
\end{aligned}$$

where $[\![\varphi]\!]$ equals 1 if the predicate φ is true, and equals 0 otherwise. (We have used the fact that $N_{\text{invert}} = 2$ in both NMOS and CMOS.)

We can now estimate the size of the access tree:

$$N_{\text{tree}} = \sum_{k=0}^{w-1} 2^k N_{\text{node}}(k) + N_{\text{buff}}$$

$$
\begin{aligned}
&< \sum_{k=0}^{w-1} 2^k[a + b(w-k-1)] + \frac{7}{2}W \\
&= (a + bw - b)(W-1) - b[(w-2)W + 2] + \frac{7}{2}W \\
&= (7N_{\text{gate}} + \frac{37}{7}N_{\text{invert}} + \frac{7}{2})W - (3N_{\text{gate}} + \frac{9}{7}N_{\text{invert}})w \\
&\qquad - (7N_{\text{gate}} + \frac{37}{7}N_{\text{invert}}) \\
&< (7N_{\text{gate}} + \frac{37}{7}N_{\text{invert}} + \frac{7}{2})W
\end{aligned}
$$

Finally, the cost of the circuitry in a slice of the tree-register interface is

$$N_{\text{slice}} = N_{\text{gate}} + N_{\text{invert}} + N_{\text{FF}},$$

from Figure 3.6. Thus the total circuitry of the chip satisfies

$$
\begin{aligned}
N_{\text{chip}} &= N_{\text{control}} + N_{\text{tree}} + N_{\text{interf}} \\
&< W + (7N_{\text{gate}} + \frac{37}{7}N_{\text{invert}} + \frac{7}{2})W + (N_{\text{gate}} + N_{\text{invert}} + N_{\text{FF}})W \\
&= (8N_{\text{gate}} + \frac{44}{7}N_{\text{invert}} + N_{\text{FF}} + \frac{9}{2})W.
\end{aligned}
$$

Table 3.2 gives the final transistor counts for both NMOS and CMOS technologies.

Table 3.2: Transistor costs by technology

Parameter	NMOS transistors	CMOS transistors
N_{invert}	2	2
N_{gate}	3	4
N_{FF}	9	14[5]
N_{chip}	$< 51W$	$< 64W$

Recall that W must be a power of 2. Using the figures in Table 3.2, an OSM supporting 32K different objects requires between 1.6M and 2M transistors, depending upon technology choices. To support 64K objects would require twice this amount; for a given technology, this would necessitate a much larger die size and would considerably reduce yields. We therefore conclude that a 32K-object OSM is the largest

[5]For example, consider minor modifications to Figure 5.51(a) of reference [56].

that can be economically implemented using the same device density as a one-megabit DRAM.

The amount of memory spanned by a single OSM chip depends on how the chip is used. The most straightforward design is for each block of the object space to correspond to one word of memory; in this case, eight OSM chips are required to control one megabyte of object memory. However, most applications have objects whose minimum size is larger than one word. For instance, the garbage collection algorithm of section 2.1 stores a header with each object, so that a minimum object size of two words seems appropriate. In this case, only four chips are needed per megabyte; an object granularity of four words would reduce this to two OSM chips per megabyte; and so on. If desired, each "object" controlled by an OSM chip can represent a larger amount of memory, with finer granularity established under software control. Power-of-two granularities can be easily implemented by discarding, external to the OSM chip, the least significant bits of properly-aligned memory addresses.

Of course, one-megabit DRAM chips will soon disappear. Already four- and even sixteen-megabit chips are becoming prevalent. The OSM chip design scales directly with memory technology: as transistor densities permit a fourfold increase in DRAM cells per memory chip, they also permit a fourfold increase in objects per OSM chip. Table 3.3 shows the amount of memory controllable by a single OSM chip for different transistor densities and object granularities. Note that the table is indexed by the *device size* obtainable in producing DRAM chips of a given size; this should not be construed to imply that OSM chips and DRAM chips can be built with the same size devices. Rather, OSM device sizes can be expected to always be one generation behind those of DRAM chips. As a result, each entry in the table should be divided by four to estimate the amount of memory supportable by an OSM chip of the same generation as a given DRAM chip. Table 3.4 shows the ratio of OSM chips to DRAM chips of the same generation, regardless of transistor density.

3.4.3 Wire costs

The above discussion of transistor counts is a very crude tool for estimating chip size. The analysis is based upon an assumption that only up to one third of the chip space is wasted due to inefficiency of wiring. It is not necessarily clear that

Table 3.3: Object space per OSM chip, given chip density and object size

Equivalent DRAM transistor size	Object granularity 1 word	2 words	4 words
1 Mbit	128KB	256KB	512KB
4 Mbit	512KB	1MB	2MB
16 Mbit	2MB	4MB	8MB
64 Mbit	8MB	16MB	32MB

Table 3.4: Ratio of OSM chips to DRAM chips

Object granularity 1 word	2 words	4 words
4:1	2:1	1:1

this assumption is valid. Better estimates of VLSI chip sizes are generally obtained by analyzing the amount of space taken up by wires, rather than transistors. Had the crude analysis produced more positive results, it would have been important to validate those results with a more careful analysis of wire costs. Instead, it appears that efforts will be better spent in pursuing alternative designs, such as the one described in reference [39].

3.4.4 Propagation delays

There are many factors that influence the amount of delay through an inverter that are beyond the scope of this analysis. It is important, however, to consider the fanout from each gate, since a fanout of M causes approximately M times the delay of a fanout of 1 [12]. This analysis assumes that a gate with outdegree 1 experiences a propagation delay of δ time units. This approach is admittedly crude, but is sufficiently informative for the present analysis.

Clearly the critical paths in the OSM chip run through the access tree. These paths vary depending on the command being serviced. For instance, the critical path

for the Clear chip space command is the buffering of the CLEAROHB signal to all memory elements in the OHB register. The other three commands depend on the speed of propagating the SELECT signals to the leaves; in addition, the Return header address command must generate the HDR signals on the way back up the tree.

As in section 3.4.2, assume the CLEAROHB signal is propagated by using buffer gates at every third level with a fanout degree of eight. Then it requires approximately $(8/3)w\delta$ time units to distribute the CLEAROHB signal. Assuming a 32K-object OSM chip, $w = \log_2 32\text{K} = 15$. A reasonable value for δ is 1 nsec; this is conservative enough to account for wire delays as well as gate delay.[6] Under these assumptions, the Clear chip space command will complete approximately 40 nsec after it is decoded by the control unit.

The critical path for the Create object and Delete object services is the generation of the SELECT_l signal (see Figure 3.5). It is easy to see that 5δ time units are needed per node, for a total delay of $5w\delta$ time units. An additional 2δ time units are required at the tree-register interface. Using the typical values of $w = 15$ and $\delta = 1$ nsec, the time to create or delete an object is about 77 nsec. Note that the SELECT signals can be propagated down the tree while the control unit is decoding the command. The CREATEOBJ signal only takes 40 nsec to reach the OHB after decoding, so there is plenty of time to decode the command in parallel with the access tree activity.

The most time-consuming command is the Return header address service. It also requires 77 nsec to propagate the SELECT signals to the OHB register, but must additionally generate the HDR signals. The critical path in the upward direction depends on the fanout of the CLMHDR_r signal, which is used to calculate $\text{HDR}[j]_p$ for each $k < j < w$ at each level $0 \leq k < w$. At the lowest level of the tree (level $w-1$) there are no copies of the $\text{HDR}[j]_p$ signal to be generated, so the critical path runs from CLMHDR_r to CLMHDR_p with a delay of 4δ time units. At higher levels of the tree, the critical path runs from CLMHDR_r through an inverter and eventually to each copy of $\text{HDR}[j]_p$.

The delay through the HDR critical path depends somewhat on how the buffering of the CLMHDR signal is arranged. Section 3.4.2 argued for a particular buffering

[6] Recall that 0.5μ technology is assumed in this design. The value of δ decreases proportionally to device size [42], but eventually wire delays dominate gate delays [29].

scheme that required little circuitry; this scheme is not the most efficient possible for minimizing delays. Fortunately, the largest fanouts occur towards the top of the tree where there are fewer nodes; thus some circuitry can be spent to gain speed at the higher levels without significantly increasing the cost of the chip. The best buffering scheme we have discovered propagates all the HDR signals to the top of the tree in approximately 168δ time units ($\approx$ 168 nsec) for a tree of height $w = 15$. Thus the Return header address command can be completed in approximately 250 nsec.

Clearly the Return header address command is the one that limits the performance of the chip. The service time of 250 nsec is greater than the cycle time even for a one-megabit DRAM chip, and almost twice the cycle time of a four-megabit DRAM chip [16]. Since a 32K-object DRAM chip and a four-megabit DRAM chip are expected to be of the same generation, a Return header address request requires about two memory cycles to service. Other commands can be serviced within one memory cycle.

It is likely that the gap between the performances of the OSM and DRAM chips will increase as technology permits higher densities, since the number of gates on an OSM critical path will increase while that for a DRAM chip will remain relatively constant. (Delays for DRAM address decoding will increase slightly with DRAM size.) It is therefore important to consider alternative implementation methods and architectural enhancements in order to improve performance. This is the subject of the next section.

3.5 Improving performance

Obviously the propagation delay through the OSM chip is proportional to the height of the access tree, so it is important to consider how the height of the tree might be reduced. Two methods of shrinking the tree have been investigated.

The first method is to replace the single, deep access tree presented above with a number of shallower trees. Each smaller tree contains its own CAR and OHB registers, and imitates a miniature version of the OSM chip previously described; it differs from the original design only in that (1) there is a single control unit for all trees on the chip, and (2) there must be fanout and arbitration logic to distribute the

address lines to the correct tree and to determine which header address is returned to the client. The advantage of this scheme is that decoding the address lines to select a tree can be done more efficiently than the bitwise decoding that takes place inside the tree. That is, standard N-to-1 decoder logic can be used to select one of N trees to receive the SELECT signal. The primary disadvantage of the method is the additional circuitry required for the replication of the CAR register and for the fanout and arbitration logic. Because of these tradeoffs, it is likely that only a limited number of trees can be placed on a chip. Note, however, that even a small number of trees would reduce the access tree delay considerably.

Another idea is to reduce the height of the access tree by combining adjacent levels of the tree into single levels of more complex nodes. If the resulting circuits are expressed in two-level form, some reduction of the delay through the access tree can be expected; however, the increased fanout degree of certain signals mitigates this speedup somewhat. Drawbacks of this method are a small cost in transistors together with the increased layout complexity due to the use of a quaternary tree instead of a binary one. Combining three levels of the tree into one was also considered, but the complexity of the logic circuits increases dramatically at this level. It appears that the high fanouts, logic complexity, and layout difficulties outweigh any slight benefits that might be achieved by a three-to-one compression.

A final variation of interest is to combine the multiple tree approach with two-to-one level compression. This combination appears to achieve the lowest delays of any method investigated to date.

Although the OSM services requests in approximately the time required for one or two memory accesses, it is important to remember that modern processors are equipped with caches. This produces *average* memory access times below 30 nsec. If requests to the OSM are frequent enough, there is a danger that it will become a system bottleneck. This is not likely to be a problem for the intended use of the chip in the garbage collection architecture, since header lookup operations are only required when partially-copied objects or unscanned pointers are referenced. However, one can imagine pathological cases involving very large arrays of pointers where lookup operations would be more frequent. Two approaches can be followed to mitigate the possibility of performance degradation: reducing the number of lookup requests, and

providing overlapping access.

Reducing the frequency of requests comes essentially for free by taking advantage of existing memory caching hardware. By caching previously fetched memory, it is only necessary to check the header address for the *first* reference to each address lying within a partially-copied object. Subsequent references will have been cached and need not be rechecked.

Caching will result in correct behavior, provided the invariant is maintained that any heap address in the cache either is a valid reference or will not be accessed before being replaced. As described in section 2.1, the garbage collection algorithm ensures that a running process will never be allowed to reference an object in *from-space*. When a flip occurs, all descriptors held by the process are tended, causing the objects they reference to be copied into *to-space*. This makes the invariant described above easy to maintain: whenever a word of memory is copied from *from-space* to *to-space*, the memory arbiter broadcasts an invalidation request for the *to-space* address, causing all processor caches to discard their private copies.[7] Using this technique, each cached heap address always falls into one of two categories:

- The address lies in *from-space*, in which case it will not be referenced prior to the next flip.
- The address lies in *to-space*, in which case it either is valid or will be invalidated before the next attempted reference.

The other approach to increasing system performance is to design for overlapping access. Additional throughput can be achieved by using a *pipelined* design for the OSM chip. The present combinational design reserves the resources of the entire chip to one user from the time of a request until that request is satisfied. By introducing memory elements at various levels of the tree, overlapping requests could be in progress simultaneously, thus increasing throughput. However, the greater bandwidth would come at the expense of additional chip area dedicated to memory elements, and would possibly degrade the latency of individual requests as well.

[7] A superior approach to data coherence is discussed in section 6.3.3.

3.6 Other alternatives

There are a number of alternatives that have been considered in developing this design. In some cases, the benefits and drawbacks of different approaches vary only slightly from those presented above; in other cases, the alternatives had to be discarded as too costly.

Removal of CAR register. The presence of the CAR register on each OSM chip is not the only way to return header addresses for objects that span more than one chip space. An alternative approach may be implemented as follows. If a chip contains the address supplied by the client, but does not have any header to the left of that address within its chip space, it sends a signal to the chip containing the previous segment of the object space, requesting it to respond. If this chip also does not have the header, it requests its neighbor to respond, and so on. All chips still process the request in parallel, so the only additional cost of this operation is the delay incurred in passing the signal from one chip to the next as many times as necessary. As each chip controls a fairly large space, only very large objects will involve more than one such propagation.

There are several reasons why this approach was not used in the design presented here. First, its performance degrades quickly if the multiple-tree-per-chip design described in section 3.5 is used. In this case, the number of trees containing portions of large objects increases. Finding the header of such objects then incurs a delay proportional to the number of trees involved, rather than just the number of chips. Second, the propagation of signals from one chip to the next incurs more delay than keeping the CAR on-chip, because of the extra time needed to get a signal onto or off of a chip. The circuitry required to implement the CAR is minimal, and worth the time savings. Finally, the "daisy-chain" approach is inconsistent with the goal of providing constant-time performance regardless of the size of an object. Theoretically, the worst-case response time is proportional to the size of garbage-collected memory, rather than to the height of a single OSM access tree. Thus the guaranteed performance of the daisy-chain technique changes as memory is added. This is not the case for the selected design.

The removal of the CAR register in favor of daisy-chaining does have some

advantages. The most obvious drawback of these registers is that the size of the object space appears to be hard-wired into the chip design. This is not actually the case: multiple banks of OSM chips can be used to overcome this limitation. The single drawback to using multiple banks for this purpose is that objects may not span the space controlled by more than one bank. Removal of the CAR register also simplifies the circuitry implementing the Create object command, and chip area may be saved as well: as the number of trees per chip increases, the circuitry associated with each tree's registers is no longer negligible.

Removal of the Delete object command. The Delete object command is not utilized in the current memory arbiter protocol. Its original projected use in merging small slice data regions into larger ones was found to be lacking in merit, for two reasons:

- Alignment on cache-line boundaries reduces the potential gains from this method, since in general it is not acceptable to change the alignment of data along cache-line boundaries when copying it from one location to another. Thus there will usually still be wasted space between adjacent small data regions.
- The complexity of the algorithmic scheme to merge small slice data regions is too great to be feasibly implemented in hardware.

However, removing this command from the OSM does not significantly reduce the circuit size or increase the performance of the chip. In fact, the only circuitry that would be removed resides in the control unit which, as mentioned in section 3.4, is negligible. Therefore the Delete object command was not removed; it may be that future designs will find a use for it.

A more ornate command set. The OSM chip was initially envisioned to have a slightly larger command set, and to detect malicious or accidental misuse of the object space. The command set was to include a Validate address service to inform the client whether or not a given address was contained within any object. This would allow, for example, quick detection of processes that accidentally attempted to dereference invalid heap pointers. Another discarded command was the Clear subspace service, which was to permit clearing of arbitrary contiguous portions of the object space.

The primary difficulty with the Validate address command is that it requires knowing where objects end, as well as where they begin. This means that two additional W-bit registers, the *object termination bitmap (OTB)* and the *valid address bitmap (VAB)*, must be maintained. This in turn complicates the creation and deletion of objects. The Create object command must pass two addresses down the access tree, rather than just one, in order to be able to set bits in both the OHB and the OTB. This means that the SELECT circuitry must be duplicated. Furthermore, an additional INSIDE signal must be propagated so that all VAB bits between the beginning and ending addresses of the object can be set.

The Delete object command becomes particularly complex. Either the client must supply both the beginning and the ending addresses of the object to be deleted, or the OSM must locate the ending address given the beginning address. In the latter case, a CLMEND signal (similar to CLMHDR) must be propagated up the tree by each node that believes it contains the end of the object. Only after both the beginning and the end of the object have been found can the registers be updated. Thus in the worst case it may take three full passes through the access tree to delete an object. The Clear subspace command is essentially identical to Delete object with both addresses supplied, except that different error conditions were checked.

Along with these additional services, the original design called for error checking. The OSM was expected to detect creation of an object that overlapped existing objects, deletion of nonexisting objects, and clearing of subspaces or chip spaces that included partial objects. These error signals were generated at the tree-register interface and OR'ed together up the access tree.

Synchronization between OSM chips was also more difficult in the original design. Since the client was viewed as untrustworthy, it was necessary to have all cooperating chips verify that no error had been detected before updating any register contents.[8] This was to be done with wired-OR logic similar to that used for the $\overline{\text{ACK}}$ signal in section 3.2. Each chip broadcast its error status on a common line, which it then monitored. If any chip reported an error, all chips responded with a $\overline{\text{NACK}}$ signal

[8]For example, suppose an object were created that spanned two chip spaces. If the second chip detected object overlap, but the first chip detected no error, the first chip would have to be instructed not to record the object's existence in its registers.

along another common line, to which the client also listened. Otherwise all affected chips updated their registers.

An early analysis of the more ornate design showed it to be unworkable due to the enormous amount of chip space required. This analysis estimated that at least 256 transistors would be required per block of object space, meaning that only 8K objects could be controlled per chip. This made the design simply too expensive to implement.

3.7 Conclusions and future work

This work has shown the feasibility of a custom VLSI chip to support an object view of memory, primarily for the purposes of real-time garbage collection. The design has been shown to scale well as chip densities increase, so that the relative cost of the OSM with respect to that of a DRAM chip remains fairly constant. Depending on the granularity of the object space, between one and four OSM chips are required to support memory equivalent to one DRAM chip of the same generation. (Lower costs can be achieved with coarser granularities.) Each service provided by the OSM chip executes in time varying between one and two memory accesses. The effect of the longer delays for the Return header address service, which may cause performance problems in pathological cases, can be mitigated through the use of caching and (possibly) pipelining techniques.

While the design presented here is *feasible*, it is not necessarily *practical*. The relative cost of OSM circuitry to DRAM circuitry is too high to justify for most applications. The primary lessons learned from this research are twofold:

(1) Some functionality in the OSM design must be sacrificed to obtain low fabrication costs.

(2) Circuit design must be very regular in order to simplify the analysis of wire routing, and to borrow from existing expertise in fabrication of standardized components.

Recently, these lessons have been applied in a preliminary redesign of the OSM that uses existing DRAM technology to greatly reduce the amount of custom circuitry.

An overview of the new design appears in reference [39]. The results from this latest effort are very encouraging, and demonstrate that the OSM chip can be manufactured at a reasonable cost.

4. A PROTOTYPE COMPILER IMPLEMENTATION

The remainder of this dissertation focuses on empirical evidence collected to determine the efficacy of the proposed hardware. Since the most pertinent results can be obtained only by testing using realistic workloads, it was necessary to modify an existing compiler to generate code targeted to the garbage-collection hardware.

Our compiler is based on version 1.37.1 of the GNU C++ compiler, developed and distributed by the Free Software Foundation [46, 53]. There are a number of reasons for selecting C++ as the source language for our experiments. Chief among these is that C++ provides both implementation efficiency and an object-oriented, type-safe programming paradigm. These two features will be necessary for the next generation of real-time systems [47], which will require adequate tools for construction of large, complex, and dynamic systems that execute efficiently. Additionally, C++ is a strongly typed language, which eases the task of providing the garbage collection module with the locations of heap pointers within allocated objects (although special consideration is required for unions). Other researchers [18] are investigating extensions to the C++ language to support real-time scheduling.

Another strong point in favor of C++ is the existence of a high-quality, retargetable compiler with source code available for modification. A number of different back ends have been developed for the GNU compilers; by targeting a GNU compiler to the garbage-collection hardware, it becomes relatively easy to port the compiler to different mutator CPUs.

Although C++ was not originally designed to support garbage collection, many users have requested that such facilities be provided. Optional implementation of garbage collection is still under consideration by the ANSI committee responsible for the C++ standard. A number of researchers have added various types of garbage collection facilities to C++; see for example [3, 6, 9, 13].

Choices of target CPUs for this research were limited primarily by the availability of tools. There are very few processors for which both a GNU back-end and an architecture simulator are available in the public domain. This project utilized the hypothetical DLX processor of Hennessy and Patterson [16], which is intended to be representative of contemporary RISC processors. (The DLX instruction set is in fact a subset of that of the MIPS R3000 [21] family of processors.) A processor simulator [17] was also available in the public domain, and was modified in a straightforward fashion to also simulate instruction and data caches, a memory bus, standard memory modules, and the garbage-collected memory module. The readability of the DLX assembly language was also a great help while debugging the compiler.

4.1 The C++ compiler

The modification of the C++ compiler to support the garbage-collection architecture was a large and interesting project. In some cases, issues raised in the compiler design caused modifications in the design of the garbage-collection module. This section describes the major issues and design decisions in the compiler implementation.

4.1.1 The arbiter interface

Recall from section 2.3 that the *memory arbiter* is the hardware module residing between the system bus and the garbage-collection processor, and having communication interfaces with each of these. The arbiter manages low-level requests to garbage-collected memory from both the mutator CPU and the garbage-collection processor, arbitrating between them when necessary. In general, requests from the mutator CPU have higher priority than those from the garbage-collection processor, and the latter have been designed to have fine-grained interruptability in order to minimize mutator stalls due to garbage-collection activities.

The mutator communicates with the memory arbiter by writing to and reading from memory-mapped ports within the arbiter. Although the arbiter has a single communication channel with the system bus, the least significant address bits are used internally to distinguish between different ports. Each writable port corresponds to a

different class of service request. (The alternative scheme of using a common port for all services requires an extra memory cycle per request to identify the service desired.) Services that return values do so by placing them in readable ports where they are read by the mutator. The arbiter port organization is illustrated in Figure 4.1.

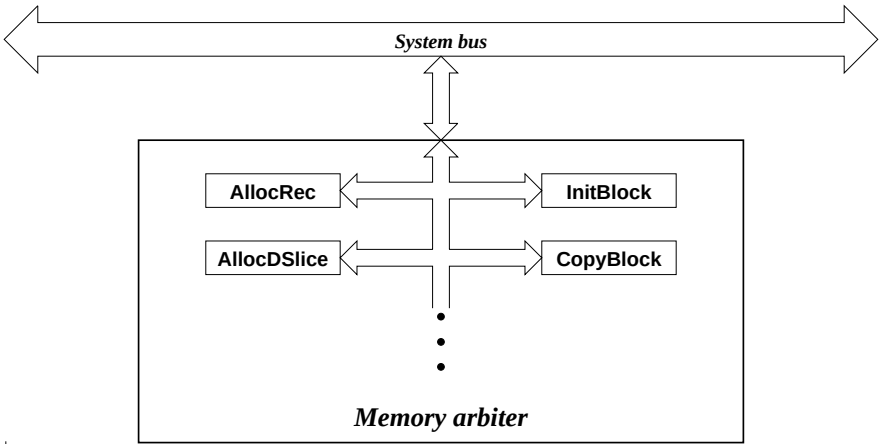

Figure 4.1: Arbiter ports

The arbiter supports seventeen mutator services and an additional twelve services for use by the collector. Five of the mutator services are unused by the C++ compiler described in this chapter. Four of these are unused because C++ does not provide slice objects, i.e., objects such as Icon [14] strings on which "internal" garbage collection can be performed.[1] The other service, `TagRead`, is of no use to the present implementation. The services used by the compiler are briefly described below, using C++-style declarations to indicate the arguments and return values of each service. Services requiring multiple arguments are invoked by writing the arguments to the same service port during consecutive memory cycles.

`word TendDesc(word desc)`

Tend a single descriptor, returning its updated value.

[1] Chapter 7 of this dissertation describes an implementation of extensions to the C++ language that support slice objects.

`void TendingDone()`

Signal to the arbiter that tending of descriptors is complete.

`void InitBlock(word addr, word Tags, int n)`

Initialize `n` words starting at `addr` to zero. `Tags` is a 32-bit mask with one bit for each of `n` words (`n` must be no greater than 32). The descriptor tag for each of the initialized words is set according to the corresponding bit of the `Tags` argument.

`void CopyBlock(word src, word dest, int n)`

Copy `n` words of memory with accompanying descriptor tags from `src` to `dest`. Requires that all copied words reside within a single object.

`void StackPush(word stack, word Tags, int n)`

Increase the size of the stack based at `stack` by `n`, initializing each of the stack-allocated words to zero. `Tags` is used as in `InitBlock`, above.

`void StackPop(word stack, int n)`

Shrink the stack based at `stack` by `n` words.

`void CopyPush(word src, word stack, int n)`

Copy `n` words of memory with accompanying descriptor tags from `src` onto the `stack`, expanding the `stack` as each word is copied. Requires that all pushed words reside within a single object.

`word AllocRec(int n)`

Allocate a record of size `n` bytes, returning a pointer to the new record.

`word AllocRecInit(int n, word Tags)`

Allocate a record of size $n \leq 128$ bytes, returning a pointer to the new record. Descriptor tags associated with each of the allocated words are initialized according to `Tags`, which is encoded as in the `InitBlock` operation.

`word AllocStack(int n)`

Allocate a stack with room to hold `n` bytes of data, returning a pointer to the first of the allocated words.

`word allocDSlice(int n)`

Allocate `n` bytes of slice region data and a slice object that refers to the slice

region data. Return a pointer to the slice object, which is flagged as potentially referring to descriptor data.

`word allocTSlice(int n)`

Allocate `n` bytes of slice region data and a slice object that refers to the slice region data. Return a pointer to the slice object, which is flagged as referring only to terminal data.

`word allocDSubSlice(word start, int len)`

Requires that `start` refers to a slice region with at least `len` bytes following `start`. Allocate a slice object that points to this memory. Return a pointer to the slice object, which is flagged as potentially referring to descriptor data.

`word allocTSubSlice(word start, int len)`

Requires that `start` refers to a slice region with at least `len` bytes following `start`. Allocate a slice object that points to this memory. Return a pointer to the slice object, which is flagged as referring only to terminal data.

`word WordRead(word addr)`

Read a single word from memory location `addr`.

`word WordWrite(word addr, word value)`

Write `value` to memory location `addr`.

Note that the `WordRead` and `WordWrite` services are not invoked by the compiler in the same manner as the other services. Instead, normal reads and writes on the system bus that refer to garbage-collected memory are intercepted by addressing hardware at the memory arbiter interface and translated into the appropriate operations on arbiter ports.

The arbiter requires the mutator process to refrain from any additional requests until an outstanding request has been completed. For those services declared above as returning `void`, the arbiter indicates completion of a request by writing a status value into the `GCStatus` port. A `GCStatus` value of zero indicates that the current request has not yet been fully serviced, while a value of 1 permits the mutator to proceed.

For allocation requests, the arbiter returns the address of the newly allocated object in the `GCResult` register. A `GCResult` value of zero indicates to the mutator

that the object has not yet been allocated. In this case, the mutator must check the `GCStatus` register to determine the reason. If the `GCStatus` register contains a value of 1, the request has simply not completed. If, however, the `GCStatus` register is zero, the mutator understands that *to-space* has been exhausted and a flip must take place. The mutator is then expected to tend its descriptors and reissue the allocation request.[2]

The `AllocRecInit`, `CopyBlock`, and `CopyPush` services were all added to the arbiter design during the course of the compiler implementation. It was quickly discovered that excessive bus traffic is generated by allocating an object and then immediately initializing it. Since the vast majority of objects are no larger than 32 words, the common case can be made fast by combining the `AllocRec` and `InitBlock` services into the `AllocRecInit` service. Larger objects must still use separate allocation and initialization requests.

Similarly, it was discovered that copying large objects causes a great deal of unnecessary bus traffic. When both the source and target objects are in the heap, the data words and the tag bits can be copied internally much more efficiently than by using the system bus and mutator registers. The `CopyBlock` and `CopyPush` services are used for this purpose when the target object is a record or a stack, respectively. (See section 4.1.4.1 for details on stack operations.) Because of time limitations, the compiler does not yet make use of the `CopyPush` primitive; instead, the `CopyBlock` service is used after the needed stack space has been reserved.[3]

4.1.2 The virtual machine

The DLX architecture is representative of a modern RISC processor. It utilizes a small orthogonal instruction set with a single base-register-plus-offset addressing mode. The DLX processor contains thirty-two general purpose registers and another thirty-two floating-point registers. Instructions pass through a five-stage pipeline, and a single branch delay slot is used. Additional information on the DLX processor

[2] An alternate protocol is discussed in section 6.3.2.

[3] In fact, the results presented in chapter 6 indicate that stack objects should be discarded, so the `CopyPush` service will also disappear.

and instruction set may be found in reference [16].

Because the algorithm employed by the garbage-collection module is exact rather than conservative, it is not safe[4] for raw data to be stored in a register designated as a tended descriptor. That is, each tended descriptor must always contain either zero or the address of an object in the heap. Since the virtual DLX machine for the original GNU back-end makes no distinction between data registers and address registers, significant changes were necessary to segregate raw data from descriptors.

Table 4.1 lists the registers in the DLX register set and their uses by the original compiler and by the modified compiler. A number of registers maintain their original uses in the modified compiler: register r0 always contains zero; r14 contains the current stack pointer; r28 is reserved to contain the address of a structure returned from a function call; r30 contains the current frame pointer; and r31 is reserved for the return address for function calls. Two registers are called upon to serve special purposes for garbage collection: r29 is reserved to point at the base of the run-time stack, and r26 points at the base of the `gcdata` object (see below). Whereas scalar values were originally returned from functions in registers r1 and r2 (if necessary), the modified compiler uses these only for raw data values; if a pointer is returned from a function, it is passed by way of r27. Finally, the remaining unused registers have been divided evenly in the new compiler between data registers (r3–r13) and address registers (r15–r25).[5]

The tended descriptors in the modified design are simply those registers that can contain addresses that point into the heap. These are the stack pointer, the address registers, the `gcdata` base pointer, the pointer return register, the returned structure address register, the stack base pointer, and the frame pointer. The link address register always points to code in C++, and thus is not a tended descriptor.

---

[4]If raw data is mistakenly interpreted as the address of an object, it is possible for space containing data that would otherwise be garbage not to be reclaimed. This in turn could violate the collector's guarantee that sufficient memory will always be available for live data, and possibly lead to failure of the program.

[5]It should be emphasized that the techniques described in this section were used in a "quick-and-dirty" port of the GNU C++ compiler to the proposed architecture. Certainly an aggressive optimizing compiler is capable of producing tighter code and making better use of registers than this prototype compiler does.

Table 4.1: Register usage in the two C++ compilers

Register	Original usage	Modified usage	TD
r0	Zero register	Zero register	No
r1–r2	Returned values	Returned scalars	No
r3–r13	General purpose registers	Data registers	No
r14	Current stack pointer	Current stack pointer	Yes
r15–r25	General purpose registers	Address registers	Yes
r26	General purpose register	`gcdata` base pointer	Yes
r27	General purpose register	Returned pointers	Yes
r28	Returned structure address	Returned structure address	Yes
r29	General purpose register	Stack base pointer	Yes
r30	Current frame pointer	Current frame pointer	Yes
r31	Link address	Link address	No

Since C++ allows explicit casting between pointers and integers, it is possible for a correct C++ program to be unsafe with respect to execution on the garbage-collected architecture. The modified compiler detects when conversions occur between pointers and integers and prints warning messages about each such occurrence. It is up to the user to ensure that no integers are incorrectly interpreted as pointers because of this sort of casting. In general, it is safe to cast from a pointer to an integer, although if no other copies of the pointer are maintained, the object pointed to may be reclaimed while the address still resides in an integer register. It is not recommended to cast from integers to pointers. It is *completely* unsafe to cast from a pointer to an integer and then back to a pointer, since the pointer could be tended in the meantime; in this case the resulting pointer would contain an illegal *from-space* address.

The original compiler back-end specifies that all parameters for DLX be passed on the stack, rather than in registers. This policy was maintained in the modified compiler for fairness in experimental comparisons.

4.1.3 Pointer location descriptions

The largest change to the compiler involved keeping track of where pointers reside within objects. This information is needed when a new object is allocated, when an argument is pushed onto the run-time stack, when the filescope and static variables are allocated at program initialization, and whenever a union within an object receives a new value. When any of these events occur, code generated by the compiler informs the garbage collector about the tag bit values for the new or modified object.

With each basic type or user-defined class, the modified compiler associates a data structure called a *pointer location description*, or *PLD*. PLDs are attached as an additional field in the syntax tree node for each class, and are themselves represented as structures in syntax tree form to allow them to be easily assembled by preexisting code in the compiler. The PLD structure is given in Figure 4.2.

```
struct pld {
  unsigned int nbits;      /* number of bits in ptrmap */
  unsigned int nwords;     /* number of words in ptrmap */
  unsigned int ptrmap[];   /* tag bits */
};
```

Figure 4.2: PLD structure

The `ptrmap` field of the PLD structure contains the tag bits associated with each object of the given class. Each bit corresponds to one word of the object, and is set to one if and only if the object contains a pointer at the indicated location. The `nbits` field indicates how many bits of the `ptrmap` field are meaningful. The `nwords` field gives the length of `ptrmap` in 32-bit words. Although this information is redundant given `nbits`, it is stored with the PLD structure to speed operations on `ptrmap`.

When the compiler determines that objects of a given class will be allocated within the heap, it generates assembly code for the associated PLD and tags it with a name uniquely determined by the hexadecimal values of the `nbits` and `ptrmap` fields. For example, allocating a six-word object containing pointers in the second

and sixth word would cause the compiler to generate the assembly code shown in Figure 4.3. PLDs with larger pointer maps have correspondingly longer names. Only the first five words of the `ptrmap` field are used in determining assembly names; beyond this, an additional serial number is assigned to distinguish duplicates.

```
__pld_6_22:
        .word   6
        .word   1
        .word   34
```

Figure 4.3: Example of PLD assembly code

The purpose of this naming scheme is to ensure that assembly code for a PLD is generated only once per compilation unit, regardless of the number of times the compiler finds that an object of the same class will be allocated. PLDs are compiled only on demand, preventing assembly code for PLDs of objects that never appear in the heap from cluttering up the object code. Additionally, if two different classes have identical PLDs, they will share the assembly code for their common PLD through this naming scheme.

Since PLDs are stored as part of the syntax tree for each class, it is straightforward to build PLDs for an aggregate class by catenating the PLDs of its components. Inheritance between classes is handled in the same fashion. Unions, however, present more complexity. When all of the components of a union have the same PLD, that PLD can be statically assigned to the union class; but when this is not the case, the tag bits for the union must be set at run time whenever the actual type of the union changes. In this latter situation, the compiler assigns an initial PLD of no pointers to the union class, with the knowledge that the run-time library (see below) will assign the correct PLD whenever an object of that class is assigned a value. The run-time system also handles the scenario wherein an aggregate object containing a union as a subobject is assigned a value; again the `CopyBlock` or `CopyPush` service is used to correctly set the tag bits for the object.

Since union members may be of any type, it is necessary for the compiler to detect all assignments not only to union members, but also to members of classes

```
struct ExStruct {
  float f;
  char c[10];
};

class ExClass {
  int i;
  double d;
  ExStruct es;
};

union ExUnion {
  ExClass ec;
  int *arrptrint[5];
};

main()
{
  ExUnion *euptr;
   ...
  euptr->ec.es.f = 31.75;
   ...
}
```

Figure 4.4: Example of union member assignment

and structs that are themselves contained in union members. Consider for example the program fragment in Figure 4.4. An idealized syntax tree for the lvalue in the assignment statement in this fragment is shown in Figure 4.5. In this figure, ovals represent syntax tree nodes, while the rectangles and triangles represent attributes of the nodes (type and name attributes, respectively). Every syntax tree node carries its type with it. Note that the "." operator is represented as a `component_ref` node whose children are the record containing the field and the field itself. Also note that the "`->`" operator is represented as such a `component_ref` node whose left child is an `indirect_ref` node, denoting a pointer dereferencing operation to find the record

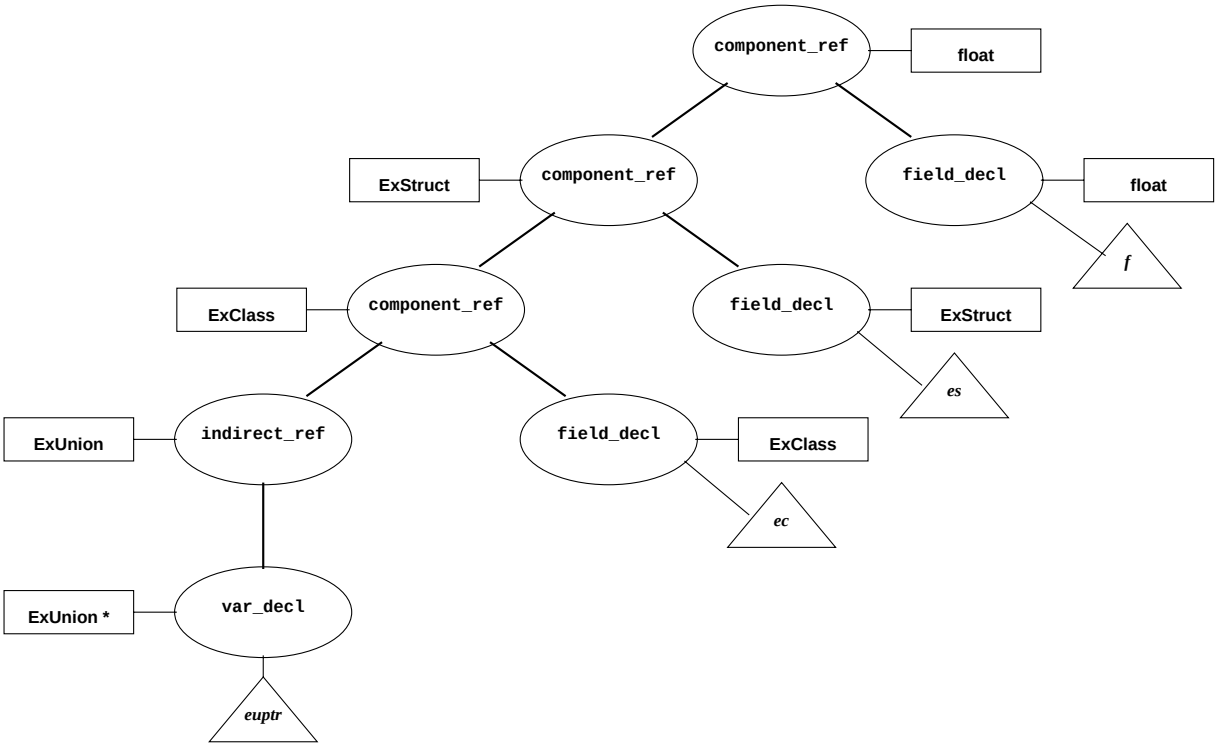

Figure 4.5: Syntax tree for lvalue of union assignment

containing the member denoted by the right child. Declared variables and fields are represented by `var_decl` and `field_decl` nodes.

This example represents one of two cases the compiler must consider when looking for assignments to portions of a named union. Beginning at the root of the syntax tree, the compiler traces down the chain of `component_ref` nodes. If any of the `component_ref` nodes has a union type that does not have a static PLD, the compiler knows that the tag bits must be initialized before the assignment takes place. In this example, none of the `component_ref`s is of union type, but the chain culminates in an `indirect_ref` that is of union type. Again, the tag bits must be initialized prior to the assignment. In general, if the chain of `component_ref`s culminates in *any* node having a union type, the tag bits associated with the effective address of that node must be updated prior to performing the assignment. If the chain of `component_ref`s is interrupted by a node that is not a `component_ref` and that is not of union type,

the compiler ceases looking for a union member initialization.

As mentioned above, PLDs are generally static objects, determined for each class at compile time. There is, however, one C++ construct that precludes static PLD generation. In the code fragment shown in Figure 4.6, a new array object must be allocated whose PLD consists of `bar+1` copies of the PLD of class `Baz`. Since `bar` is a function parameter, the size of the new PLD must be determined dynamically. A run-time library routine is used to generate PLDs for array objects of nonconstant size.

```
class Baz;
void foo (int bar)
{
  Baz *bazptr = new Baz[bar+1];
  ...
}
```

Figure 4.6: Example requiring a dynamic PLD

Array objects require one other special consideration. Section 5.7 of the ANSI base C++ language definition [11] permits array pointer arithmetic to legally generate a pointer to "the first location beyond the high end of the array." Such arithmetic is common in loops operating on array objects. It is conceivable, particularly when using optimizing compilers, that the only existing pointer to an array may step off the end of the array object in this sanctioned fashion. At such a time, the garbage collector would be within its rights to reclaim the array as garbage, since there are no live pointers that point within the array object. Furthermore, the object physically following the array object, if any, would have a live pointer to it; if this is the only live pointer to the following object, the object that follows would be incorrectly retained as live memory. To avoid these unacceptable occurrences, the compiler automatically enlarges all array objects by one element. This extra element is not seen by the programmer, and exists only to prevent accidental reclamation of a live array.

4.1.4 Special data objects

It is important that the number of tended descriptors be kept small, and that common operations be made as efficient as possible. To meet these goals, two special objects are created during each program execution.

4.1.4.1 The run-time stack object. During program execution, the run-time stack, which contains function call parameters, saved registers, local variables, and temporary storage, may contain a large number of pointers into the heap. If each such pointer were treated as a tended descriptor, the worst-case time to tend all descriptors at the time of a flip would be unacceptably large. Instead, a single tended descriptor holds the base of the stack. When this descriptor is tended, the stack contents are automatically scanned by the garbage collector.

There are two obvious ways to organize the stack: as a chain of activation frames individually allocated from the heap, or as a single large record containing sufficient space for the maximum expected stack growth. The latter approach was adopted for use by the version of the compiler used in the experiments described in chapter 5.[6] As discussed in section 2.2, a special stack object is supported by the collector hardware for this purpose. Stack objects maintain a field pointing to the current top of stack, so that only that portion of the stack that contains pertinent data is scanned for additional pointers. Stack allocation is generally believed to be more efficient than heap allocation, since heap allocation causes garbage collection to occur more frequently and requires more overhead each time a stack frame is pushed. Indeed, this belief is supported by the results of the experiments in chapter 6 (although ultimately a *variant* of heap allocation turns out to be superior).

Before the user `main()` code is executed, prologue code allocates a run-time stack object from the collector. The compiler then builds code fragments to communicate with the collector each time a stack frame is pushed or popped. Each function call causes two `StackPush` operations: one for the calling function's parameter list, and one for the activation frame of the called function.[7] At function exit, the value of the

[6] Chapter 6 investigates the relative performances of different stack organizations.

[7] The overhead of two `StackPush` calls results from following the original GNU

Caller	Callee
Pushes parameters	Sets frame pointer
Adjusts stack pointer	Pushes needed registers
	Pushes space for local variables
	Pushes space for temporaries
	Adjusts stack pointer
	Initializes local variables

Figure 4.7: Function call protocol

function (if any) is returned in one or more registers, and two `StackPop` calls mirror the `StackPush` calls invoked at function entry. The function call protocol is shown in Figure 4.7.

The modified compiler implements pushes and pops using the `StackPush` and `StackPop` arbiter services. The `StackPush` operation requires three parameters: the base of the stack object, the size of the frame to be allocated, and the tag bits for each word in the frame. The compiler generates the tag bits by catenating the PLDs of all objects being pushed at the same time. For instance, when a function call is encountered during compilation, the compiler finds the type of each parameter to be pushed, extracts the PLD from the syntax tree nodes for those types, and catenates them to produce the tag bits for the parameter list. The `StackPush` service is then invoked prior to pushing the actual parameters onto the stack, since `StackPush` clears the affected portion of the stack object to zero.

Each called function also begins with a `StackPush` call to initialize the tag bits for saved registers, local variables, and temporaries. If a register save location is to hold the contents of a tended descriptor, its tag bit must be set to one, and otherwise to zero. The compiler calculates the tag bits for the register locations, and catenates the result with the PLDs for the types of all local variables and temporaries. The resulting PLD is stored with the function code, and serves as an argument to the

---

C++ function call code generation. Chapter 6 investigates the utility of reserving space for function call arguments in the caller's activation frame, thus eliminating one `StackPush` call per function call.

`StackPush` call whenever the function is called.

The `StackPop` call requires only one argument, the number of words to be popped. The compiler generates a `StackPop` request at the end of each function body, and following each function call. The collector services the request by updating the top-of-stack pointer associated with the stack object.

Additional effort is required for inlined functions. Inlined functions do not begin with a `StackPush` to save registers and allocate space for locals and temporaries. Instead, the internal representation of the function is inserted in the calling function at each use, and any required locals and temporaries are allocated within the enclosing function body. Thus the compiler must recognize the presence of inlined functions within an enclosing function body, and catenate the PLDs for locals and temporaries in the inlined function with the other tag bits in the enclosing activation frame.

4.1.4.2 The `gcdata` record. The lifetime of filescope and static variables in C++ is the duration of program execution. Since there may be any number of such objects, and since they may be of any type, there is theoretically no limit on the number of pointers into the heap that may be contained in them. Again, it is necessary to restrict the size of the set of tended descriptors while maintaining the ability to locate all pointers into the heap. For this purpose, all filescope and static variables that contain pointers (or unions that may contain pointers) are collected together into a large record, called the `gcdata` record. Pointers that cannot be construed as heap pointers (such as pointers to functions and methods) are excluded. Among other things, this prevents virtual function tables (which can be very large) from residing in the `gcdata` record.

As with the run-time stack object, the `gcdata` record is allocated from the collector in prologue code, prior to the execution of user `main()`. Since the `gcdata` record is mobile, the compiler refers to filescope and static variables using an offset from the `gcdata` base register; register r26 is reserved to point to the current base location of the `gcdata` record.

When the compiler detects a pointer within an object that belongs in the `gcdata` record, it generates a `.descriptor` directive (rather than the usual `.word` directive) in the assembly output. This directive is of the form `.descriptor` *address*, where

address is the initial value to be assigned to the pointer. If the initial value is the address of another object in the `gcdata` record, *address* takes the form `$gcdata$+name`, where `name` identifies the location of the addressed object. This is necessary since the `gcdata` record's location is not fixed. The information in `.descriptor` directives is used by the linker while generating the `gcdata` record (see section 4.2 for more details).

4.1.4.3 Note on permanent objects. The run-time stack object and the `gcdata` record share two important properties: both have lifetimes lasting the duration of program execution, and both may be quite large. The garbage collection activity required to copy both of these objects into *to-space* at each flip may be considered to be "pure overhead," since it is certain that they will never become garbage and therefore could be given a fixed home. An early hypothesis was that there might be significant gains if these two objects were stored in a separate memory area outside of the two semispaces, but still under the control of the collector. At each flip, the permanent objects would be scanned, but not copied. In the real workloads studied in chapter 5, however, it was found that the size of the `gcdata` record is generally relatively small. Furthermore, the stack depth tends to be fairly small in "most" programs, so the amount of copying done for the run-time stack is unlikely to heavily impact the overall performance of the collector. It would seem that permanent "set-aside" memory would only be cost-beneficial if additional long-lived objects were allowed to gravitate there. Thus more measurable gains might be made by modifying the garbage collection algorithm to use generational scavenging [30, 54].

4.1.5 The run-time library

The compiler creates code to communicate with the garbage collector by generating calls to a small set of run-time library routines. There are separate routines for each of the collector services described in section 4.1.1, except for the `TendDesc` and `TendingDone` primitives. These are used only inside the subroutine `__gc_flip_spaces`, which is called whenever it is time for a flip.

Most of the library routines require arguments. For the run-time library, the normal practice of passing parameters on the stack is not appropriate. Not only is

this method too inefficient for operations that occur so frequently, but there is also a problem of circularity: if a call to a library routine to request a `StackPush` service were to push its arguments on the stack, another `StackPush` would be required to inform the collector about the arguments! Thus, a special call protocol is used for the run-time library. Each library routine expects its arguments in specific hard registers, and the compiler generates code to place the arguments there. The library routines use only a small set of "clobberable" registers, which are not guaranteed to maintain their values over function calls.

For those functions that require an array of tag bits as an argument, two library routines are provided. One of these handles the general case in which the object may be of any length, while the other deals with the common case where the object is no longer than 32 words, and therefore has only one word of tag bits. This removes unnecessary loop overhead from the vast majority of operations. The compiler automatically generates code to call the proper library routine, depending on the size of the object.

Different allocation routines are provided for different object types; for example, the routine to allocate a record is different from the one used to allocate the run-time stack object. In all cases, if the object cannot be allocated due to lack of remaining space in *to-space*, the collector returns a special status code indicating that the mutator should tend its descriptors. Code in the library routines recognizes when this has occurred, and automatically tends the tended descriptors. To reduce bus traffic and collector overhead, only those tended descriptor registers with nonzero contents are tended.

4.1.6 Optimizations

In most cases, the revised compiler produces reasonably good code without undue effort. However, there are a few constructs for which the "naturally" produced code is markedly inferior to what it should be. For these cases, specific optimizations have been developed, only one of which is discussed in this section. Another optimization that is primarily of use in certain alternative function call mechanisms is discussed in chapter 6.

The optimization discussed here requires some explanation of the concept of

"mode" employed within GNU C++. Each object type is assigned a mode that is carried around with the syntax tree for that type. The mode indicates what sort of registers may be used to hold values of that mode. For example, `SImode` objects may be stored in any register capable of holding a single integer; `PSImode` objects have the same size as `SImode` objects but are understood to contain pointers; `DFmode` objects contain double-precision floating-point values and thus require a pair of floating-point registers. The designation `BLKmode` is used for those objects that are not able to reside entirely in registers. Thus `BLKmode` is used for most record types, for example; individual members of the structure may have their own register-capable modes, but the structure as a whole is `BLKmode`. Copying an entity that is of mode `BLKmode` is always performed via a block copy.

The unmodified GNU C++ compiler uses a heuristic to determine the best way to perform a block copy. If the object to be copied is shorter than a configurable threshold value, the object is copied, one word at a time, through registers. Otherwise, the compiler generates a call to the library routine `bcopy()` to perform the copy in as efficient a manner as possible.

Neither of these methods turns out to be acceptable in the garbage-collecting compiler. Moving data one word at a time through registers would work fine, provided that `PSImode` registers are used for pointers and not for anything else. However, the original compiler has discarded all type information by the time it sets up the block copy, and it is not easy to modify this code to retain the type information without affecting large portions of the compiler. The `bcopy()` alternative is totally unacceptable, since such library routines treat the data to be copied as an untyped block of bytes. So the garbage-collecting compiler implements all block copies between heap-allocated objects by invoking the `CopyBlock` arbiter primitive.

This method works well for most data types, since `BLKmode` is intended to be used for structures, unions, and classes that are too large to fit in a single register or register pair. The compiler recognizes structures that do fit in registers and assigns appropriate register modes to them. However, in the case when a class has a constructor, instances of that class are *always* declared to be `BLKmode`. Thus if a class contains only a single word of private data per instance, but the class has a constructor, the compiler will generate an expensive `CopyBlock` call to move an instance of

that class when a simple register move would do!

The original compiler does have a reason for forcing instances of every class with constructors to have mode `BLKmode`: when allocating storage, the compiler guarantees to place a `BLKmode` object in memory, rather than a register. This ensures that a `BLKmode` object always has an address, which is required for an instance of a class with constructors, since the implicit first argument of the constructor call must be the address of the space allocated for the new object. The designers of the original compiler used `BLKmode` as a shortcut to ensure that constructed objects are addressable, even when declared as local variables that might otherwise fit in a register.

To circumvent the problem of performing `CopyBlock` calls when register moves are sufficient, the modified compiler detects all classes that contain one word or less of data per instance and that have a constructor. With each such class, it stores not only the mode normally assigned to it (i.e., `BLKmode`) but also the "original" mode of the word of data, which is defined to be `PSImode` if the data consists of a pointer and `SImode` otherwise. Whenever an instance of such a class is to be copied, the compiler uses this original mode to determine whether to copy it through an address register or a data register. Retaining the `BLKmode` designation as the "true" mode still ensures that instances of these classes are always addressable.

It would be feasible to extend this technique to larger classes with constructors if it were shown to be beneficial. For instance, it might be faster to copy a two- or four-word object through registers rather than using a `CopyBlock` call, although this is by no means assured. This would require storing, with the syntax tree node for the class, the original modes for as many words of data as are present in an instance of the class. Each word of data would then be copied through an address or data register as appropriate. The complexity of implementing this for more than a single word was not deemed worthwhile for this first implementation.

4.1.7 Compatibility considerations

Programs that use garbage collection have no need to explicitly reclaim storage. In the original GNU C++ compiler, the syntax "`delete x;`" is translated into destructor calls for the class of which `x` is an instance and any base classes of that

```
class ExClass {
private:
  int *iptr;
  char *cptr;
public:
  ...
  void *operator new(size_t);
};

void *ExClass::operator new(size_t n)
{
  ExClass *temp = (ExClass *)new char[sizeof(ExClass)];
  ...
  return temp;
}
```

Figure 4.8: Incompatibility arising from `operator new`

class, followed by a call to `builtin_delete()`, which reclaims the storage used by `x`. The revised compiler excises the call to `builtin_delete()` from the syntax tree representation of the `delete` statement, while retaining the calls to the destructors for the class and its base classes. Thus it is not necessary to rewrite programs that explicitly delete storage.

Of course, if programs have eschewed the standard `new` and `delete` mechanisms of the language in favor of their authors' favorite allocation methods, they will not be automatically compatible with the garbage collector. A common incompatibility arises when a programmer wishes to write his or her own `operator new` for a class. In this case, the programmer is responsible for allocating the storage for an instance of the class within the `operator new` function body. However, the programmer cannot simply invoke `new` on the class, since this will result in a call to `operator new`, causing an infinite recursion. The common programmer solution to this problem is shown in Figure 4.8. Here the programmer has used `new` to allocate an array of characters equal in size to an instance of the example class `ExClass`, and has then used a cast to convert the address of this array into a pointer to `ExClass`. Now,

```
class ExClass {
private:
  int *iptr;
  char *cptr;
public:
  ...
  void *operator new(size_t);
};

class DmyClass {
  int *iptr;
  char *cptr;
};

void *ExClass::operator new(size_t n)
{
  ExClass *temp = (ExClass *)new DmyClass;
  ...
  return temp;
}
```

Figure 4.9: A workaround for the `operator new` problem

the compiler has allocated an array of characters, so it has set the tag bits for the entire array to zero. But `ExClass` contains pointers. When the next flip occurs, the pointers within this object will not be tended and will thus still point into *from-space*. Eventually *from-space* will be initialized to zeroes, destroying the data addressed by these pointers.

One solution to this problem is shown in Figure 4.9. Here a class `DmyClass` has been created with the same nonstatic data members as `ExClass`. `DmyClass` has no overloaded `operator new`, so `ExClass::operator new()` can legitimately perform a `new DmyClass` without problem. Since an instance of `DmyClass` and an instance of `ExClass` have pointers in the same locations, it is safe to cast a `DmyClass` pointer into an `ExClass` pointer. One drawback of this workaround is that it requires care by the programmer to ensure that the dummy class does indeed have exactly the

same pointer locations as the original class. This can be particularly difficult in the presence of inheritance. It would be beneficial to add a warning message in the compiler that complains whenever a pointer cast is made between two types that do not have the same PLD.

A better solution, not yet implemented here, would be to recognize constructs such as the one depicted in Figure 4.8 and give them slightly different semantics. Whenever the compiler detects that an array of characters is being allocated, but the resulting pointer to this array is immediately cast to a pointer to another type `T`, the compiler can generate code to allocate an object from the collector whose tag bits are set according to the PLD of type `T`. A similar technique could be used to intercept and translate invocations of `::operator new`, which might also be used by a programmer to allocate untyped memory.

4.1.8 Limitations

Because of limited resources, certain bugs in the compiler have not been fixed. These have not proven to be important in compiling real programs. For example, version 1.37.1 is one of the first versions of the GNU C++ compiler to incorporate multiple inheritance, and predictably is not perfect. In particular, virtual base classes are not correctly implemented. If a program that uses virtual base classes under multiple inheritance is to be compiled with this compiler, the `virtual` specifier must be removed. In those cases where the sharing of virtual base classes is necessary to the semantics of the program, additional work is required to correctly compile the program.

Currently the modified compiler can only be used without the optimization flag. There are only a small number of things that break when the optimization flag is turned on, but they require a fair amount of effort to fix. Future studies may be done to compare the original and modified compilers when full optimization is turned on. However, the unmodified GNU C++ 1.37.1 optimizer breaks code even when targeted, for example, to the SPARC architecture, so such studies seem unlikely to be fruitful.

One bug still persists from implementing the optimization for small `BLKmode` objects discussed in section 4.1.6. When the type of a conditional expression (i.e.,

an expression involving the `?:` operator) is one of these small `BLKmode` objects, the compiler is currently unable to generate correct code. The fix for this is a little tricky, and because of time limitations it has not yet been implemented. This is a mild irritant, but expressions involving the conditional operator can be rewritten as if-then statements until this is repaired.

One final feature that has not yet been implemented involves spilling register contents into the stack. When too many `register` specifiers are declared in a function, the compiler will run out of hard registers and eventually will decide to allocate some register variables to stack slots. This decision comes much later in the compilation process than when the PLDs for the stack frames are created. To incorporate register spillage would require some tricky backpatching of PLDs, which has not yet seemed worthwhile. A workaround is simply to remove enough of the `register` specifiers from the source program that the register spillage does not occur; the end result is the same.

4.2 The linker and librarian

C++ supports separate compilation of program fragments. In an actual execution environment, this requires linking separate object modules together to form an executable program image. Similarly, the DLX assembly code files produced by the original and modified G++ compilers must be linked together to satisfy external references, eliminate collisions between local labels, add prologue and epilogue code, and construct certain global data structures. The `dlxln` catenating linker, supported by the `dlxlib` assembly code librarian, serves this purpose. Both of these tools were developed locally.

The linker supports two modes of linking to target either the standard DLX architecture or the enhanced garbage-collection architecture. The primary function of the linker, supported by both modes, is to catenate all needed modules together into one file, keeping a running list of unresolved external references. Since the compiler uses a uniform labeling scheme for code branch points in each assembly file it generates (`L1`, `L2`, and so forth), the linker must also renumber all of these labels.

The programs used in the experiments of chapters 5 through 7 were originally

written to run under the UNIX[8] BSD 4.3 operating system. They are therefore heavily dependent on the standard UNIX run-time libraries. The `dlxlib` librarian was created to allow `dlxln` to extract assembly code modules from libraries similarly to the way the UNIX linker extracts object modules from object libraries. Those library routines needed to complete the workload programs have been compiled for DLX and collected into common libraries. When a library is specified on the `dlxln` command line, the linker searches that library for modules that contain any currently unresolved references. Dependencies between modules within a library are all resolved in one pass.

The ANSI base C++ language definition [11] states that "initialization of nonlocal static objects in a translation unit is done before the first use of any function or object defined in that translation unit." For each computation unit, the GNU C++ compiler outputs a function containing code to initialize all such objects, together with a directive indicating to the linker that this function is to be executed prior to invoking the user `main()` function. Similarly, if any nonlocal static objects in a compilation unit require destructing, the compiler generates a single function containing code to call the destructors for all such objects. `dlxln` creates two data structures, `___CTOR_LIST___` and `___DTOR_LIST___`, each of which consists of a null-terminated list of function addresses. All compilation-unit initialization and destruction functions require no arguments and return no results. Hand-coded DLX prologue and epilogue code ensures that these functions are called at the correct times.

In addition to the foregoing, more work is required to link programs targeted to the garbage-collection architecture. This is primarily because of the `gcdata` record. The linker must find all declarations of objects that are to be placed in the `gcdata` record (identified by the compiler with a preceding `.gcdata` directive) and collect them into a single record. All code references to symbols associated with such objects must be replaced with their assigned offsets within the `gcdata` record. The linker must also generate a PLD describing the entire `gcdata` record, which is used by the prologue code to allocate it from the collector.

Finally, the linker must handle initialization of objects in the `gcdata` region. The

[8] UNIX is a registered trademark of AT&T Bell Laboratories.

compiler generates normal directives for static initializations, indicating compile-time values to be stored directly in the indicated locations. Since these locations are not known for the `gcdata` record until after it has been allocated from the collector, assembly directives do not suffice. The linker generates a subroutine containing DLX assembly code to perform initialization of all objects in the `gcdata` record that have static initializers. This subroutine is called by the prologue code after the `gcdata` record has been allocated.

The prologue code also allocates the run-time stack object. The linker reserves a word of storage to contain the size of the stack, which defaults to 16K. The user can set this value using a command-line option, or modify it by hand in the simulator before execution begins.

4.3 The `dlxsimgc` simulator

The `dlxsimgc` simulator, written primarily by Dr. Kelvin Nilsen, emulates the overall machine architecture under study, including DLX processor, instruction and data caches, memory bus, standard memory module, and the proposed garbage-collected memory module. This simulator is based on the original DLX processor simulator [17], but has been extensively rewritten in C++ to permit simulation of the interactions between the processor and the other architectural components.

The simulator is organized in a modular fashion, with each major component in the system defined as a separate object. Objects communicate with each other by calling each other's public functions. The basic unit of computation is the processor cycle, under the assumption that the CPU is the fastest component in the system. At each processor cycle, the simulator's main loop calls the `doCycle()` function in each object, informing the object that it may do one processor-cycle of work. Some of the objects are not fast enough to perform any functions more often than once every several cycles; these objects simply mark time in their `doCycle()` functions until enough cycles have elapsed for them to do more work.

Since no hardware prototypes have yet been built, and since there are many design tradeoffs that are not yet well understood, the simulator has been designed to be highly tunable. Well over a hundred parameters may be individually configured

in order to test different possible hardware configurations and assumptions about operation latencies. Some of the most important of these are: the sizes and locations of standard and garbage-collected memory; instruction and data cache parameters, including line size, overall cache size, associativity, and number of write buffers; the proportional amount of garbage collection that accompanies each allocation request; the word size and alignment restrictions of the central processor; and the latencies associated with each microoperation within the garbage-collected memory module. The simulator also supports a debugging configuration, permitting interactive inspection of registers and memory, breakpoints on execution addresses, and watchpoints on memory addresses to be monitored for change.

4.3.1 Explanation of statistics

The `dlxsimgc` simulator gathers a number of statistics to be used in performance analysis studies. Examples of the simulator's statistical output may be found in the raw experimental data collected in reference [44]. Following is a short description of the meaning of each reported statistic.

`Program image ends at` *address*
: The combined code and data size of the simulated program may be found by subtracting 0x100 from *address*.

`total machine instructions executed`
: The number of instructions actually passing through the execute stage of the DLX pipeline.

`cycles stalled for instruction fetch`
: The number of times the DLX pipeline was stalled because the instruction to be fetched was not immediately available.

`cycles stalled for memory operations`
: The number of cycles during which the DLX pipeline was stalled during a load or store because the memory subsystem was busy.

`cycles stalled following loads`
: The number of times the DLX pipeline was stalled while waiting for values loaded on a previous instruction to become available.

`cycles stalled for floating point results`

The number of times the DLX pipeline was stalled because an instruction could not execute until the result of a previous floating-point instruction became ready.

`cycles stalled for floating point processors`

The number of times the DLX pipeline was stalled because a floating-point instruction could not be issued until a floating-point unit became available.

`cycles stalled for branch-delay instruction fetches`

The number of cycles spent waiting to fetch and begin decoding the instruction immediately following a branch instruction.

`cycles stalled for trap interfacing`

UNIX system calls are implemented as DLX traps. Prior to executing each trap, the simulator ensures that all of the mutator's write buffers have been flushed to memory. This statistic accounts for the time spent flushing these buffers.

`total machine cycles executed`

The latency of program execution, measured in processor cycles.

`total number of traps executed`

The number of system calls issued during program execution.

`Arbiter Operations` (table)

This table gives a detailed breakdown of the *latencies* and *costs* of operations requested by the CPU and performed by the arbiter. The *cost* of an operation is the number of machine cycles actually required to perform the operation. The *latency* of an operation is the number of elapsed cycles between the time the CPU finishes issuing the request and the time when the CPU claims the results of the operation (by reading from the `GCResult` or `GCStatus` registers). Thus latencies tend to be slightly larger than costs, since the CPU is rarely lucky enough to ask for the result of an operation precisely when it is ready.[9] The costs and latencies are further broken down by whether the operations took place while

[9] Another reason that latencies are higher than costs is that many operations cause cache invalidation requests to be broadcast on the bus for affected memory addresses. Time to finish invalidation requests after the operation has completed is reflected in the latency, but not in the cost.

garbage collection was in progress, since the costs of operations tend to be higher in this case. The columns in this table report: the number of invocations of a given operation; the mean value of the cost or latency; the standard deviation of the costs or latencies; and the low and high range of the costs. (The range of latencies is not gathered because it is essentially redundant given the range of costs and the mean difference between costs and latencies; it was not deemed worthwhile to slow the simulations further by gathering this statistic.)

`warning: no garbage collection activity (New =` *address*`)`

This message is produced if no flip occurred during the simulated program execution.

`number of allocations unimpeded by GC`

This statistic measures the number of allocation requests (including `allocInitRec`, `allocRec`, `allocDSlice`, `allocTSlice`, `allocDSubSlice`, `allocTSubSlice`, `allocTStack`, and `allocDStack` operations) that were issued either when garbage collection was idle or when garbage collection was sufficiently far ahead that the allocation could proceed without any additional garbage collection first taking place.

`total cycles required for GC`

The number of machine cycles during which garbage collection was active.

`bus utilization`

The percentage of total processor cycles during which the bus was in use with a read, write, or cache invalidation request.

`utilization due to cache invalidation requests`

The percentage of total processor cycles during which the bus was in use with cache invalidation requests.

`icache hit rate`

The ratio of instruction fetches that were satisfied by the cache to the total number of instruction fetches.

`dcache hit rate`

The ratio of operand fetches that were satisfied by the cache to the total number of operand fetches.

`number of executions of malloc (free)`

The number of "jump-and-link" instructions executed that targeted the address of the `malloc()` (`free()`) subroutine. This and the following four statistics are only reported if the program invokes `malloc` and/or `free`.

`total cycles for 'malloc' ('free') executions`

The number of machine cycles executed between the time of the jump-and-link instruction to `malloc()` (`free()`) and the time at which program execution returned to the caller, totalled for all calls.

`mean cycles per 'malloc' ('free') execution`

Self-explanatory.

`standard deviation of 'malloc' ('free') execution times`

Self-explanatory.

`range of malloc (free) costs`

The lowest and highest execution times for the `malloc()` (`free()`) subroutine.

`CPU usage:` *s* `system,` *u* `user`

The total number of CPU seconds of system and user time required to execute the simulation, as measured by the UNIX library routine `time()`.

4.3.2 Limitations

Although `dlxsimgc` is capable of extensive simulation of the architectural components under study, it does have a few limitations. First, it is not designed to simulate input from and output to slower system devices such as disks, tapes, printers, and terminals. Second, the programs chosen for the simulation studies reported throughout the rest of this dissertation were written originally to run on the UNIX BSD 4.3 operating system, and therefore rely on a core set of UNIX system calls. The compiler translates each system call into a numbered `trap` instruction, informing the simulator that the stack contains arguments for a particular system call to be emulated. The simulator makes no attempt to accurately gauge the costs of these calls, but merely simulates their effect and keeps a count of the number of system calls executed.

The overall effect of these limitations is *conservative* in the sense that the true costs of garbage collection will be less than those reported in the studies of the following chapters. The presence of slow devices in the system will permit garbage collection to execute while the CPU remains idle for longer periods. Similarly, no allocation requests will be made during system calls (at least in typical systems of today), so garbage collection will be able to proceed at a faster rate relative to allocations than is seen in the reported empirical results.

5. PERFORMANCE ANALYSIS

This chapter describes the design and results of experiments carried out to test the performance of the proposed garbage collection hardware. The goals of these studies were twofold: to compare the performance of a standard contemporary architecture with that of the real-time garbage collection architecture under real workloads; and to gather statistics on the requests to, and behavior of, the garbage collection architecture under real workloads. Such statistics will be useful in refining the architecture to increase performance.

The results of these studies have been most enlightening. Although the overall performance statistics show that the testbed system used in these experiments did not perform competitively against the traditional architecture, detailed analysis of the gathered statistics proves that the poor performance of the garbage collection system was due almost entirely to correctable problems with the protocol employed at the beginning and end of function calls. This chapter recounts this analysis and explains the faults in the prototype system; a description of solutions to these faults is delayed until the end of chapter 6, which describes alternative mechanisms for function calls. A thumbnail analysis at the end of chapter 6 demonstrates that slight modifications to the protocol will make the garbage-collection architecture's performance competitive with that of traditional architectures. Preliminary experiments appear to support this claim.

5.1 System definitions

The experiments described in this chapter compare a traditional RISC architecture with an identical architecture enhanced to use the real-time garbage-collected memory module. Both architectures use a single hypothetical DLX processor as

the CPU. The DLX instruction set is a subset of the MIPS R3000 [21] family of processors. The DLX processor uses a five-stage integer pipeline and a scoreboarded floating-point architecture configured with one addition unit, one multiplication unit, and one division unit. Additional information about the DLX configuration may be gleaned from the configuration parameters listed in reference [44].

Both processors are configured with on-chip instruction and data caches. Each cache is a 32-kilobyte, two-way associative cache with a line size of one word. The data cache employs a write-through policy to provide a coherent view of memory with the garbage collection architecture. Other coherence schemes involving a write-back policy are possible but were not considered in these experiments. The garbage-collection architecture is presumed to use a mirror cache or some other method to be able to quickly determine those words in *from-space* that must be invalidated at the time of a flip. In a real implementation, the mutator would be responsible for invalidating its cache, using methods such as those described in reference [34]. The issue of data coherence is treated in more detail in section 6.3.3, where a method employing a write-back cache is introduced.

Both architectures employ a standard memory bus with 32-bit-wide address and data channels. The bus is assumed to operate a standard cache-coherence protocol that supports signals to invalidate any word in memory. Each architecture is equipped with four megabytes of static-column DRAM memory; the use of static-column memory improves access times when the addresses of consecutive requests to memory occupy the same row within the DRAM chip. For the garbage-collection architecture only, a garbage-collected memory module, also employing static-column DRAM, is included as well. Configurable parameters of the garbage-collected memory module are not discussed here, but are available in reference [44] for the interested reader.

The programs executed on each simulated architecture were compiled with the GNU C++ compiler, version 1.37.1, as targeted to the DLX CPU by graduate students at Stanford University and the University of California at Berkeley. Numerous modifications were necessary to remove bugs from the DLX back-end. When targeting the garbage-collection architecture, the additional compiler modifications outlined in chapter 4 were also used.

The boundary of system testing is the system bus and the UNIX kernel interface. No attempt was made in these simulations to account for the speed of mass storage devices or the I/O paths to these devices, nor was the cost of system calls considered. None of these factors is considered to be critical in comparing the performance of these two architectures, since programs targeted to the two architectures will have essentially identical kernel requests and I/O requirements.

5.2 Parameters and factors

A *parameter* is any element or quantity that can vary and thereby affect performance. This section lists the major design parameters that can affect the performance of one or both architectures.

5.2.1 System parameters

- Use of the standard architecture versus the real-time garbage-collection architecture.
- Cache size.
- Cache associativity.
- Cache cycle time, in CPU cycles.
- Cache line size.
- Bus protocol used.
- Bus width.
- Memory cycle time, in CPU cycles.
- Organization of main memory (interleaving, pipelining, etc.).
- Time required to perform each action within the garbage-collected memory module, in CPU cycles.
- Size of the garbage-collected memory module.

- Assumed time to perform I/O to components outside the system boundary.
- Assumed time to perform kernel calls.
- Whether I/O devices outside the system boundary are assumed to use memory-mapped I/O (and therefore the resources of the system bus) or use independent I/O paths.

5.2.2 Workload parameters

- The individual program run during a given test.
- The inputs provided to the program run during a given test.

A *factor* is a parameter that is varied during experimental evaluation. Because of limited computational resources and the amount of time necessary to run test cases that adequately exercise the garbage collector, only a small number of factors could be selected for testing. The factors used in the experiments discussed in this chapter are:

- the simulated architecture;
- the program to be simulated;
- the inputs to the simulated program;
- the speed of the DLX CPU relative to the rest of the components; and
- the size of garbage-collected memory.

These factors were chosen because, of all the parameters listed above, they were considered the most likely to affect the differences in overall system performance between the two systems. In recent years it has been shown that selection of a varied workload is very important in obtaining unbiased performance figures (see, for example, the relevant discussion in reference [16]); hence the architectures have been simulated executing three very different programs on two contrasting input sets each. The workload is described in detail in section 5.3.

CPU speed was chosen as a factor because of the continually growing gap between CPU speeds and memory speeds. Since garbage-collected memory cannot always respond as quickly as traditional memory, the difference in their performance should be exacerbated as the CPU's demand on memory increases. In these experiments, performance using a "normal" CPU is contrasted with that of a "fast" CPU. The normal CPU has approximately the characteristics of contemporary RISC processors, while the fast CPU is considered to be able to perform all operations at twice the speed of the normal CPU. The speeds of other components are held constant during all trials.

The size of garbage-collected memory is important in determining the frequency of flips, and hence the fraction of time that garbage collection is active. A minimal amount of garbage-collected memory would be expected to affect overall performance more detrimentally than a large amount that requires fewer flips during execution. In the trials analyzed in this chapter, garbage-collected memory is varied between a "small" and a "large" memory size. The small memory size consists of the smallest amount of garbage-collected memory that permits the program to execute, while the large memory size contains twice that amount.

Of the remaining parameters that were not selected as factors, the cache parameters and the bus width should also be varied in future investigations. Like CPU speed, these parameters affect the rate at which demands are placed on traditional and garbage-collected memory, so their effects should be nontrivial. The variation in performance due to the change in CPU speeds may be treated as indicative of how important these other parameters might be.

5.3 Workload

Three programs were ported to the two target architectures for use in these experiments. Two quite different input sets were prepared for testing each program. The programs were selected in large part by availability, since there are few production-quality C++ programs in the public domain, but they were also selected because of their very different execution behavior. Although only one of these is a program that might be found operating today in a hard real-time domain, the programs tested

should be indicative of the complexity and performance of programs that will be used in future-generation real-time systems.

The `sfft` program, contributed by James I. Lathrop, performs a sliding discrete Fourier transform (DFT) on a file of 8-bit audio data, computing the DFT of the last n samples as each new sample arrives. The DFT of the last n samples is printed every one thousand iterations. Two sample input files, containing 512 and 2048 samples of raw data, respectively, were used as input test cases. Octal dumps of the input files are listed in reference [44]. The `sfft` program was chosen because it represents a typical real-time task, because it exercises the floating-point units more thoroughly than the other programs, and because it does *not* make use of dynamically allocated memory, in contrast to the other programs. Because of this last point, `sfft` never causes garbage collection to be initiated. This makes `sfft` valuable as a measure of the overhead of garbage collection for those tasks that do not make good use of it.

The `lisp` program, written by Timothy Budd, is a C++ implementation of the basic Lisp interpreter provided as a companion to the programming languages textbook by Kamin [20]. The first input test case used with `lisp` begins by defining the relational database functions given in the textbook, and then uses these functions to create a database and make several queries to it. The other test case implements the alpha-beta pruning method of searching minimax game trees, and tests it on two large multiway trees. These test cases are also included in reference [44]. The `lisp` program is an example of a task having small code size, high instruction and data locality, small average function size (and hence a high rate of function call invocation), and heavy utilization of dynamic memory allocation. It also never explicitly frees any dynamically allocated data.[1] Thus, when targeted to the traditional architecture, `lisp` never incurs the overhead of calls to `free()`. Including one test case that uses `free()` (see below) and another that does not permits comparison of these two allocation methods' performance with that of the garbage-collection system.

The final test program is `troff`, the basic typesetting program from the GNU `groff` package, version 1.03, written by James Clark. The first test case for `troff`

[1]The `lisp` interpreter is mainly intended for use by students in writing small introductory programs. The authors did not feel that it was worthwhile to implement storage reclamation for this environment.

consists of input for a 16-page paper, processed with no additional command line arguments. The second test case uses a 24-page paper with command line arguments specifying that three macro files should be processed before typesetting the input file. These two test cases are too large for publication, but are available upon request. The `troff` program exhibits large code size, slightly lower instruction and data locality, and a larger average function size than `lisp`. However, it also makes very heavy use of dynamic memory allocation. In contrast to `lisp`, `troff` is careful to explicitly free almost all of its allocated data.

5.4 Results of experiments

To compare the performances of the systems while varying the five factors listed above required thirty-six trials. Each of the six test cases (three programs with two input data sets each) was executed for each combination of the other three factors. There are only six such combinations, rather than the expected eight, since varying the amount of garbage-collected memory while using the traditional architecture makes no sense.

This section contains a series of tables analyzing the empirical results of different measured statistics. (The unedited empirical results may be found in reference [44].) For brevity, each of the three non-workload factors has been represented by a letter, as follows:

Level 1		**Level −1**	
Factor	**Description**	**Factor**	**Description**
A	Garbage-collected architecture	$\overline{\text{A}}$	Traditional architecture
B	Large GC memory	$\overline{\text{B}}$	Small GC memory
C	Fast mutator CPU	$\overline{\text{C}}$	Slow mutator CPU

Each of these factors has two levels, designated in the table as level -1 and level 1. Each trial in the tables that follow is designated by the letters ABC with none, some, or all of these negated with a bar, representing one of the eight possible combinations of factors. An unaltered letter A, B, or C represents level 1 of the corresponding factor, while one of these letters appearing beneath a horizontal bar represents level -1 of the corresponding factor. Thus A$\overline{\text{B}}$C designates the trial using

the garbage-collection architecture, a small garbage-collected memory size, and a fast mutator CPU. As noted above, two of the eight trials are redundant; $\overline{\mathrm{ABC}}$ is equivalent to $\overline{\mathrm{A}}\mathrm{B}\overline{\mathrm{C}}$, and $\overline{\mathrm{AB}}\mathrm{C}$ is equivalent to $\overline{\mathrm{A}}\mathrm{BC}$. The redundant trials are presented in the tables for the sake of readability, and because the redundant information is used in calculating the effects explained below.

As a rule, each statistic is analyzed using four tables. Each of the first three tables corresponds to one of the three workload programs; the statistic is summed (or averaged, as appropriate) over that program's two input sets. The fourth table provides the statistic's value for the entire combined workload. Whether the individual tables or the combined table are more instructive depends on the nature of each statistic; where the combined data is not very meaningful, this is noted in the discussion following the tables.

With each trial in a given table is recorded three numbers. The first of these is the raw data gathered as just described. The second column shows the percent increase or decrease in the statistic's value when compared with its value on the traditional architecture with a normal CPU (hereafter called the *basic configuration*). The third column compares just those test cases using the garbage-collected architecture, providing the percent increase or decrease in the statistic's value when compared with the case where a normal CPU and a "large" garbage-collected memory is used. This is referred to in later discussions as the *standard garbage-collection configuration.*

To the right of the trial data in each table is a calculation of the importance of each of the factors and their interactions. These *effects* are calculated using a standard nonlinear regression model; this discussion of the model summarizes that of Jain [19].[2] For each factor $f \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$, define a variable x_f such that $x_f = i$ if factor f is set to level i. For each subset $X \subseteq \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$, define q_X to be the *effect* of the combined factors in X. Thus q_{A} is the effect of factor A, and q_{BC} is the combined effect of factors B and C. Now, if y is an observed value of the statistic being measured with the factors set to levels x_{A}, x_{B}, and x_{C}, the following model

[2]The analysis here presumes that three factors are present, but the same methods are pertinent for any number of factors.

relates y to the effects of the various factors and their interactions:

$$y = q_{\emptyset} + x_{\mathrm{A}} q_{\mathrm{A}} + x_{\mathrm{B}} q_{\mathrm{B}} + x_{\mathrm{C}} q_{\mathrm{C}} + x_{\mathrm{AB}} q_{\mathrm{AB}} + x_{\mathrm{AC}} q_{\mathrm{AC}} + x_{\mathrm{BC}} q_{\mathrm{BC}} + x_{\mathrm{ABC}} q_{\mathrm{ABC}}$$

The effects are calculated by substituting the results of the eight trials into this model and solving for the q_X's.

One characteristic of this model is that $q_{\emptyset}$ represents the mean value of the statistic being analyzed. Thus it is represented in the tables as μ. Each other q_X is represented simply as X.

If the statistic y is measured to have values $y_1, \ldots, y_n$ over n trials, with mean value μ, then the *total variation* of y is calculated as

$$V = \sum_{i=1}^{n} (y_i - \mu)^2$$

Using the regression model for effects described above, it can be shown that (for $n = 8$)

$$V = nq_{\mathrm{A}}^2 + nq_{\mathrm{B}}^2 + nq_{\mathrm{C}}^2 + nq_{\mathrm{AB}}^2 + nq_{\mathrm{AC}}^2 + nq_{\mathrm{BC}}^2 + nq_{\mathrm{ABC}}^2$$

Here nq_{A}^2, for example, represents the portion of the total variation that is explained by factor A alone. In the tables that follow, the variation explained by each factor or combination of factors is presented as a percentage of the total variation. This can be used as a measure of which factors have an impact on each statistic y.

Finally, beneath each table is a measure of correlation between the measured statistic and the elapsed CPU cycles for the corresponding test cases. (Elapsed CPU cycles are analyzed in section 5.4.1.) The correlation coefficient ρ is a real value between -1.0 and +1.0. A correlation coefficient approaching 1.0 indicates a direct relationship between the compared quantities, while a value near -1.0 indicates an inverse relationship. A value of 0.0 means that the two quantities are completely unrelated. Values close to none of 0.0, 1.0, and -1.0 generally do not give much information about any relationship between the quantities and should be interpreted with care, particularly considering the small number of trials over which the correlation coefficient is being computed.

5.4.1 Elapsed CPU cycles

This statistic represents the overall performance of the systems under comparison. It is immediately apparent from tables 5.1–5.4 that the garbage-collected memory architecture is not performing competitively with the traditional architecture. The standard garbage-collected configuration is slower than the basic configuration by 45% for `sfft`, 174% for `troff`, and a whopping 428% for `lisp`. The average change in performance over all test cases is +199%, indicating an execution rate one-third that of the traditional architecture. Clearly there is a serious problem with the implementation of the system tested here. An explanation of this performance decrease is obtained during the analysis of the remaining statistics in this section.

Note that the values for the cases using the fast CPU must be interpreted carefully. Since the CPU speed is doubled with respect to the other components, the number of CPU cycles increases greatly due to increased pressure on the memory subsystems. However, the actual latency of execution decreases. This is shown in the following set of statistics, where execution latency is measured in terms of the number of cycles of a normal CPU; that is, the number of CPU cycles is halved for those test cases using a fast CPU.

In the present analysis, it is useful to contrast the increase in memory cycles when the traditional architecture is upgraded to a faster CPU with the increase when the garbage-collected architecture is thus modified. Note that when the larger garbage-collected memory is used, the decreased performance is roughly similar between the two systems, although it is uniformly higher for the garbage-collected architecture. As garbage-collected memory decreases, the effect of increased CPU speed on overall performance becomes more pronounced. Thus it is important to ensure that sufficient garbage-collected memory is available to avoid this effect.

The variation in performance among the trials is primarily explained by the choice of architecture, with the CPU speed and the interaction between these two factors as secondary considerations. In light of the above discussion, this comes as no surprise.

Table 5.1: Elapsed CPU cycles, `sfft`

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	494	+0%		μ	690	
$\overline{\text{AB}}$C	614	+24%		A	136	70.79%
$\overline{\text{A}}$B$\overline{\text{C}}$	494	+0%		B	0	0.00%
$\overline{\text{A}}$BC	614	+24%		C	84	27.01%
A$\overline{\text{BC}}$	718	+45%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	932	+89%	+30%	AC	24	2.20%
AB$\overline{\text{C}}$	718	+45%	+0%	BC	0	0.00%
ABC	932	+89%	+30%	ABC	0	0.00%

Correlation with `sfft` performance: $\rho = +1.000$

Table 5.2: Elapsed CPU cycles, `lisp`

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	524	+0%		μ	2111	
$\overline{\text{AB}}$C	701	+34%		A	1499	88.87%
$\overline{\text{A}}$B$\overline{\text{C}}$	524	+0%		B	-91	0.33%
$\overline{\text{A}}$BC	701	+34%		C	404	6.46%
A$\overline{\text{BC}}$	3013	+475%	+9%	AB	-91	0.33%
A$\overline{\text{B}}$C	4572	+773%	+65%	AC	316	3.95%
AB$\overline{\text{C}}$	2767	+428%	+0%	BC	-30	0.04%
ABC	4088	+680%	+48%	ABC	-30	0.04%

Correlation with `lisp` performance: $\rho = +1.000$

Table 5.3: Elapsed CPU cycles, `troff`

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	1782	+0%		μ	4348	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	2596	+46%		A	2159	81.25%
$\overline{\text{AB}}\overline{\text{C}}$	1782	+0%		B	-225	0.88%
$\overline{\text{A}}\text{BC}$	2596	+46%		C	867	13.10%
A$\overline{\text{BC}}$	5481	+208%	+12%	AB	-225	0.88%
A$\overline{\text{B}}$C	8435	+373%	+73%	AC	460	3.69%
AB$\overline{\text{C}}$	4881	+174%	+0%	BC	-75	0.10%
ABC	7233	+306%	+48%	ABC	-75	0.10%

Correlation with `troff` performance: $\rho = +1.000$

Table 5.4: Elapsed CPU cycles, all experiments

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	2800	+0%		μ	7149	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	3911	+40%		A	3793	84.21%
$\overline{\text{AB}}\overline{\text{C}}$	2800	+0%		B	-317	0.59%
$\overline{\text{A}}\text{BC}$	3911	+40%		C	1355	10.75%
A$\overline{\text{BC}}$	9212	+229%	+10%	AB	-317	0.59%
A$\overline{\text{B}}$C	13939	+398%	+67%	AC	799	3.74%
AB$\overline{\text{C}}$	8365	+199%	+0%	BC	-105	0.06%
ABC	12253	+338%	+46%	ABC	-105	0.06%

Correlation with overall performance: $\rho = +1.000$

5.4.2 Execution latencies

As discussed above, this statistic is identical with the previous one, except that the values for the fast CPU cases have been cut in half. This provides a comparison in terms of real clock time. The primary purpose in showing Tables 5.5–5.8 is to demonstrate the change in the fraction of variation explained by each factor. In terms of clock time, the effect of the faster CPU is much lower in the `lisp` and `troff` cases than in the previous set of tables, while it is much higher in the case of `sfft`. It is clear that `sfft` is affected much less than the other workloads by the presence of garbage collection, even though for `sfft` garbage collection is pure overhead, since it makes no explicit use of dynamic memory allocation. This phenomenon is explained in the analysis of the next statistic.

In the remainder of this section, statistics quantified in terms of elapsed time will only be analyzed in terms of elapsed cycles, as in section 5.4.1. However, the reader should remain aware of the possible differences between a CPU cycle analysis and a clock time analysis.

Table 5.5: Total latencies, `sfft` (normal CPU cycles)

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	494	+0%		μ	496	
$\overline{\text{A}}$BC	307	-38%		A	96	42.72%
$\overline{\text{AB}}$$\overline{\text{C}}$	494	+0%		B	0	0.00%
$\overline{\text{A}}$BC	307	-38%		C	-110	56.09%
A$\overline{\text{BC}}$	718	+45%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	466	-6%	-35%	AC	-16	1.19%
AB$\overline{\text{C}}$	718	+45%	+0%	BC	0	0.00%
ABC	466	-6%	-35%	ABC	0	0.00%

Correlation with `sfft` performance: $\rho = +0.145$

Table 5.6: Total latencies, `lisp` (normal CPU cycles)

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	524	+0%		μ	1483	
$\overline{\text{A}}$B$\overline{\text{C}}$	351	-33%		A	1045	93.40%
$\overline{\text{A}}\overline{\text{B}}$C	524	+0%		B	-61	0.32%
$\overline{\text{A}}$BC	351	-33%		C	-225	4.33%
A$\overline{\text{BC}}$	3013	+475%	+9%	AB	-61	0.32%
A$\overline{\text{B}}$C	2286	+336%	-17%	AC	-138	1.63%
AB$\overline{\text{C}}$	2767	+428%	+0%	BC	1	0.00%
ABC	2044	+290%	-26%	ABC	1	0.00%

Correlation with `lisp` performance: $\rho = +0.839$

Table 5.7: Total latencies, `troff` (normal CPU cycles)

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	1782	+0%		μ	3045	
$\overline{\text{A}}$B$\overline{\text{C}}$	1298	-27%		A	1505	89.21%
$\overline{\text{A}}\overline{\text{B}}$C	1782	+0%		B	-150	0.89%
$\overline{\text{A}}$BC	1298	-27%		C	-437	7.52%
A$\overline{\text{BC}}$	5481	+208%	+12%	AB	-150	0.89%
A$\overline{\text{B}}$C	4218	+137%	-14%	AC	-195	1.50%
AB$\overline{\text{C}}$	4881	+174%	+0%	BC	0	0.00%
ABC	3617	+103%	-26%	ABC	0	0.00%

Correlation with `troff` performance: $\rho = +0.746$

Table 5.8: Total latencies, all experiments (normal CPU cycles)

Trial	cycles ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	2800	+0%		μ	5023	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	1955	-30%		A	2645	89.66%
$\overline{\text{AB}}\text{C}$	2800	+0%		B	-211	0.57%
$\overline{\text{A}}\text{BC}$	1955	-30%		C	-772	7.64%
A$\overline{\text{BC}}$	9212	+229%	+10%	AB	-211	0.57%
A$\overline{\text{B}}$C	6969	+149%	-17%	AC	-349	1.56%
AB$\overline{\text{C}}$	8365	+199%	+0%	BC	1	0.00%
ABC	6126	+119%	-27%	ABC	1	0.00%

Correlation with overall performance: $\rho = +0.766$

5.4.3 CPU instructions executed

Tables 5.9–5.12 show the total number of CPU instructions executed. Note the very high correlation between this statistic and the total number of elapsed CPU cycles. This might seem at first to be a trivial correlation; but in fact it is highly indicative of the source of the poor performance observed in section 5.4.1. The code executed by the two machines is generally identical, *except* when the CPU must communicate with the arbiter as part of the garbage-collection protocol. (Different code is also produced for the operators `new` and `delete`; but fewer instructions are required to implement these operators for the garbage-collection architecture than for the traditional architecture, so this cannot be the source of the problem.) Thus the source of the overhead is in the run-time library that implements this protocol.

A certain amount of overhead in the run-time library is inevitable. The performance loss discovered here, however, is much higher than originally expected. Indeed, the total instructions executed for all experiments is 207% higher when garbage collection is used than when it is not; the number of elapsed CPU cycles is only 199% higher. This appears to indicate that the garbage-collection architecture would be very competitive with a traditional architecture if the protocol overhead problem were overcome. As described in chapter 6, it seems likely that most of this overhead

Table 5.9: Total instructions executed, `sfft`

Trial	instructions ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	304	+0%		μ	335	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	304	+0%		A	31	96.78%
$\overline{\text{A}}\text{B}\overline{\text{C}}$	304	+0%		B	0	0.00%
$\overline{\text{A}}\text{BC}$	304	+0%		C	4	1.61%
A$\overline{\text{BC}}$	358	+18%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	375	+23%	+5%	AC	4	1.61%
AB$\overline{\text{C}}$	358	+18%	+0%	BC	0	0.00%
ABC	375	+23%	+5%	ABC	0	0.00%

Correlation with `sfft` performance: $\rho = +0.916$

Table 5.10: Total instructions executed, `lisp`

Trial	instructions ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	335	+0%		μ	825	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	335	+0%		A	490	96.12%
$\overline{\text{A}}\text{B}\overline{\text{C}}$	335	+0%		B	-9	0.03%
$\overline{\text{A}}\text{BC}$	335	+0%		C	69	1.91%
A$\overline{\text{BC}}$	1191	+256%	+2%	AB	-9	0.03%
A$\overline{\text{B}}$C	1476	+341%	+27%	AC	69	1.91%
AB$\overline{\text{C}}$	1164	+247%	+0%	BC	-3	0.00%
ABC	1428	+326%	+23%	ABC	-3	0.00%

Correlation with `lisp` performance: $\rho = +0.989$

Table 5.11: Total instructions executed, `troff`

Trial	instructions ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	710	+0%		μ	1383	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	710	+0%		A	673	96.04%
$\overline{\text{A}}\text{BC}$	710	+0%		B	-17	0.06%
$\overline{\text{A}}\text{BC}$	710	+0%		C	95	1.91%
A$\overline{\text{BC}}$	1888	+166%	+3%	AB	-17	0.06%
A$\overline{\text{B}}$C	2289	+222%	+24%	AC	95	1.91%
AB$\overline{\text{C}}$	1841	+159%	+0%	BC	-5	0.01%
ABC	2202	+210%	+20%	ABC	-5	0.01%

Correlation with `troff` performance: $\rho = +0.965$

Table 5.12: Number of instructions executed, all experiments

Trial	instructions ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	1349	+0%		μ	2543	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	1349	+0%		A	1194	96.10%
$\overline{\text{A}}\text{BC}$	1349	+0%		B	-26	0.05%
$\overline{\text{A}}\text{BC}$	1349	+0%		C	168	1.90%
A$\overline{\text{BC}}$	3438	+155%	+2%	AB	-26	0.05%
A$\overline{\text{B}}$C	4140	+207%	+23%	AC	168	1.90%
AB$\overline{\text{C}}$	3363	+149%	+0%	BC	-7	0.00%
ABC	4006	+197%	+19%	ABC	-7	0.00%

Correlation with overall performance: $\rho = +0.975$

can indeed be erased with a more careful protocol implementation.

It turns out that almost all of the overhead can be attributed to the code generated for function calls. As described in chapter 4, two `StackPush` operations and two `StackPop` operations take place for each function call. The following table shows the dominance of the stack manipulation operations among all of the arbiter invocations. Well over 99% of all requests to the arbiter are pushes or pops of the activation stack. Also shown in the table is the ratio of pops to pushes, which gives an indication of the average size of an activation frame or parameter block; one `StackPush` is required for each 32 words or fraction of 32 words pushed, while a single `StackPop` suffices for any number of words popped. The data in the table is taken from the raw data for the experiments performed with the standard garbage-collection configuration (AB$\overline{\text{C}}$).

Table 5.13: Breakdown of arbiter calls

Program	`StackPush` invocations	`StackPop` invocations	Other invocations	Stack fraction	Pops/Pushes
`sfft`	824,742	678,582	1217	0.9992	0.8228
`lisp`	14,487,488	14,470,416	48,940	0.9983	0.9988
`troff`	17,734,191	15,689,027	126,943	0.9962	0.8847
Total	33,046,421	30,838,025	177,100	0.9972	0.9332

Each `StackPush` operation is executed by a run-time library routine that executes $I_{\text{push}} = 12 + 4n$ instructions for some integer n that depends on the reaction time of the arbiter. This is because the mutator executes a busy-waiting loop, continually checking for completion of the operation. Each `StackPop` operation similarly requires $I_{\text{pop}} = 10 + 4n$ instructions. The following table calculates a rough estimate of the cost of the stack manipulation operations for each of the test programs. To do this, the value of n used in the definition of I_{push} and I_{pop} must be estimated. In order to be conservative, the smallest latencies for the `StackPush` and `StackPop` operations have been taken from the raw data for these test cases in reference [44]. These latencies (in CPU cycles) were then divided by the cycles per instruction (CPI) for these test cases, calculated from the statistics in this section and in section 5.4.1 and shown in the table below. This gives a rough figure for $4n$, which is then rounded

up to the nearest integral multiple of 4 and used to calculate I_{push} and I_{pop} according to the formulas given above. Using these values, the total number of instructions spent in manipulating the stack is estimated as I_{stack}, and the percentage of total instructions dedicated to stack manipulation is also estimated.

Table 5.14: Cost of stack manipulation

Program	CPI	I_{push}	I_{pop}	I_{stack}	% total instructions
`sfft`	2.004	48	18	51,802,092	13.80%
`lisp`	2.378	32	18	724,067,104	49.05%
`troff`	2.651	40	18	991,770,126	43.33%
Total	2.488	40	18	1,876,941,290	45.34%

It must be emphasized that the figures in this table are very rough and should only be relied upon to indicate the general nature of the problem. Even so, these results show clearly that the cost of stack manipulation explains the vast majority of the lost performance due to garbage collection. Additionally, the final column explains why `sfft` suffers far less performance degradation from garbage collection than do `lisp` and `troff`. Clearly the average amount of computation performed per function call in `sfft` is much higher than in the other two programs, with the result that `sfft` is impacted far less by the cost of stack manipulation.

5.4.4 Allocation latencies

This statistic measures the total time (in CPU cycles) spent by each program performing allocation. For the traditional architecture, this is measured by recording the number of cycles spent executing the functions `malloc()` and `free()`, while for the garbage-collection architecture it represents the sum of the latencies of all `AllocInitRec`, `AllocRec`, `InitBlock`, and `AllocDStack` operations. It should be noted that the data for `sfft` is essentially meaningless, since only one allocation (to obtain an I/O buffer) takes place during any execution of that program. The reader should also be reminded that `lisp` makes no use of `free()`.

For the other programs, it is apparent from examining tables 5.15–5.18 that

Table 5.15: Total latencies for allocations, `sfft`

Trial	cycles ($\times 10^3$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	13	+0%		μ	19	
$\overline{\text{A}}$B$\overline{\text{C}}$	22	+69%		A	2	16.00%
$\overline{\text{AB}}\overline{\text{C}}$	13	+0%		B	1	4.00%
$\overline{\text{A}}$BC	22	+69%		C	4	64.00%
A$\overline{\text{BC}}$	18	+38%	+0%	AB	1	4.00%
A$\overline{\text{B}}$C	19	+46%	+6%	AC	-1	4.00%
AB$\overline{\text{C}}$	18	+38%	+0%	BC	1	4.00%
ABC	29	+123%	+61%	ABC	1	4.00%

Correlation with `sfft` performance: $\rho = +0.677$

Table 5.16: Total latencies for allocations, `lisp`

Trial	cycles ($\times 10^3$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	12773	+0%		μ	9249	
$\overline{\text{A}}$B$\overline{\text{C}}$	18612	+46%		A	-6443	90.14%
$\overline{\text{AB}}\overline{\text{C}}$	12773	+0%		B	-56	0.01%
$\overline{\text{A}}$BC	18612	+46%		C	1828	7.26%
A$\overline{\text{BC}}$	2166	-83%	+10%	AB	-56	0.01%
A$\overline{\text{B}}$C	3669	-71%	+86%	AC	-1092	2.59%
AB$\overline{\text{C}}$	1974	-85%	+0%	BC	-8	0.00%
ABC	3415	-73%	+73%	ABC	-8	0.00%

Correlation with `lisp` performance: $\rho = -0.857$

Table 5.17: Total latencies for allocations, `troff`

Trial	cycles ($\times 10^3$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	50031	+0%		μ	34689	
$\overline{\text{AB}}$C	73806	+48%		A	-27230	91.10%
$\overline{\text{AB}}\overline{\text{C}}$	50031	+0%		B	-122	0.00%
$\overline{\text{A}}$BC	73806	+48%		C	6878	5.81%
A$\overline{\text{BC}}$	5738	-89%	+5%	AB	-122	0.00%
A$\overline{\text{B}}$C	9666	-81%	+78%	AC	-5009	3.08%
AB$\overline{\text{C}}$	5442	-89%	+0%	BC	-48	0.00%
ABC	8989	-82%	+65%	ABC	-48	0.00%

Correlation with `troff` performance: ρ = -0.806

Table 5.18: Total latencies for allocations, all experiments

Trial	cycles ($\times 10^3$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	62817	+0%		μ	43959	
$\overline{\text{AB}}$C	92440	+47%		A	-33670	90.92%
$\overline{\text{AB}}\overline{\text{C}}$	62817	+0%		B	-177	0.00%
$\overline{\text{A}}$BC	92440	+47%		C	8711	6.09%
A$\overline{\text{BC}}$	7923	-87%	+7%	AB	-177	0.00%
A$\overline{\text{B}}$C	13364	-79%	+80%	AC	-6101	2.99%
AB$\overline{\text{C}}$	7435	-88%	+0%	BC	-55	0.00%
ABC	12433	-80%	+67%	ABC	-55	0.00%

Correlation with overall performance: ρ = -0.827

allocations can be performed much more quickly using the garbage-collection architecture than using `malloc()` and `free()`. In fact, on average the garbage-collection system allocates objects 88% more quickly than the traditional architecture. This lends credence to the hypothesis that the garbage-collection architecture can perform competitively with traditional architectures, provided that the function call overhead problem is solved.

If the amount of garbage-collected memory available is minimal, the program suffers a small decrease in allocation performance, but the result is still much better than the traditional architecture's performance. More important is the effect of using a faster mutator processor. As CPU speed increases, the cost of allocation rises more quickly than does the overall increase in elapsed cycles. This is true for both the traditional and the garbage-collection architecture. Thus the difference in allocation latencies between the two architectures becomes more important as processor speed increases.

5.4.5 Cache performance

This section details the statistical behavior of the instruction and data caches as the several factors are varied. In order to better explain the hit rates achieved by each cache, tables showing the number of hits, the number of fetches, and the overall hit rate have been computed for each program.

The statistics for the instruction cache are slightly anomalous, reflecting a different instruction fetch method than was intended to be simulated. The DLX CPU uses a delayed-branch mechanism with a single branch slot; that is, whenever a branch instruction is executed, the following instruction is also executed regardless of whether the branch is taken. This gives the CPU time to determine the next instruction to execute and to begin fetching it without incurring a stall. A bug in the simulator[3] causes the second instruction following a branch instruction to be fetched as well,

[3]This bug was discovered too late to permit rerunning all of the test cases. Given the very limited computing resources available, and given the many weeks necessary to run all of these simulations, the decision was made to be satisfied with the present results. The effect of this bug on performance differences between the two architectures is thought to be insignificant.

regardless of whether it will be executed. Thus this fetch is often wasted. This explains the fact that the number of instructions fetched (Tables 5.20–5.29) is uniformly larger than the number of instructions executed (Tables 5.9–5.12). The effect of this on instruction cache hit rates is uncertain; but it is probable that the hit rates reported here are slightly high, since sequential instructions are more likely to be in the instruction cache than are branch targets.

In all cases, instruction cache hit rates are slightly higher when the garbage-collection architecture is selected. The effect of this can be seen most clearly for `troff`, whose code size is sufficiently large that the effects of cache line replacement are apparent. In this case the garbage-collection architecture exhibits hit rates that are 3–4% higher than those of the traditional architecture. Unfortunately, this is probably due to the excess stack manipulation overhead. Since the programs spend a great deal of their time in busy waiting loops, repeatedly fetching these instructions artificially inflates the instruction cache hit rates. Correcting the stack manipulation problem will likely reduce the hit rates closer to those of the traditional architecture.

Table 5.19: Instruction cache hits, `sfft`

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{A}}\overline{\text{B}}\overline{\text{C}}$	317	+0%		μ	356	
$\overline{\text{A}}\overline{\text{B}}$C	317	+0%		A	39	95.48%
$\overline{\text{A}}$B$\overline{\text{C}}$	317	+0%		B	0	0.00%
$\overline{\text{A}}$BC	317	+0%		C	6	2.26%
A$\overline{\text{B}}\overline{\text{C}}$	384	+21%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	406	+28%	+6%	AC	6	2.26%
AB$\overline{\text{C}}$	384	+21%	+0%	BC	0	0.00%
ABC	406	+28%	+6%	ABC	0	0.00%

Correlation with `sfft` performance: $\rho = +0.918$

Table 5.20: Instruction cache fetches, `sfft`

Trial	fetches ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	317	+0%		μ	356	
$\overline{\text{AB}}$C	317	+0%		A	39	95.48%
$\overline{\text{A}}$B$\overline{\text{C}}$	317	+0%		B	0	0.00%
$\overline{\text{A}}$BC	317	+0%		C	6	2.26%
A$\overline{\text{BC}}$	384	+21%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	406	+28%	+6%	AC	6	2.26%
AB$\overline{\text{C}}$	384	+21%	+0%	BC	0	0.00%
ABC	406	+28%	+6%	ABC	0	0.00%

Correlation with `sfft` performance: $\rho = +0.918$

Table 5.21: Instruction cache hit rate, `sfft`

Trial	hit rate	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.99996	+0%		μ	0.99996	
$\overline{\text{AB}}$C	0.99996	+0%		A	0.00000	33.33%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.99996	+0%		B	0.00000	0.00%
$\overline{\text{A}}$BC	0.99996	+0%		C	0.00000	33.33%
A$\overline{\text{BC}}$	0.99996	+0%	+0%	AB	0.00000	0.00%
A$\overline{\text{B}}$C	0.99997	+0%	+0%	AC	0.00000	33.33%
AB$\overline{\text{C}}$	0.99996	+0%	+0%	BC	0.00000	0.00%
ABC	0.99997	+0%	+0%	ABC	0.00000	0.00%

Correlation with `sfft` performance: $\rho = +0.870$

Table 5.22: Instruction cache hits, `lisp`

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	373	+0%		μ	980	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	373	+0%		A	607	96.06%
$\overline{\text{A}}\text{BC}$	373	+0%		B	-12	0.04%
A$\overline{\text{BC}}$	373	+0%		C	86	1.93%
A$\overline{\text{B}}\text{C}$	1432	+284%	+3%	AB	-12	0.04%
AB$\overline{\text{C}}$	1787	+379%	+28%	AC	86	1.93%
ABC	1397	+275%	+0%	BC	-3	0.00%
	1728	+363%	+24%	ABC	-3	0.00%

Correlation with `lisp` performance: $\rho = +0.989$

Table 5.23: Instruction cache fetches, `lisp`

Trial	fetches ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	374	+0%		μ	980	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	374	+0%		A	606	96.05%
$\overline{\text{A}}\text{BC}$	374	+0%		B	-12	0.04%
A$\overline{\text{BC}}$	374	+0%		C	86	1.93%
A$\overline{\text{B}}\text{C}$	1432	+283%	+3%	AB	-12	0.04%
AB$\overline{\text{C}}$	1787	+378%	+28%	AC	86	1.93%
ABC	1397	+274%	+0%	BC	-3	0.00%
	1728	+362%	+24%	ABC	-3	0.00%

Correlation with `lisp` performance: $\rho = +0.989$

Table 5.24: Instruction cache hit rate, `lisp`

Trial	hit rate	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.9996	+0%		μ	0.9997	
$\overline{\text{A}}$BC	0.9996	+0%		A	0.0001	92.59%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.9996	+0%		B	0.0000	0.00%
$\overline{\text{A}}$BC	0.9996	+0%		C	0.0000	3.70%
A$\overline{\text{BC}}$	0.9998	+0%	+0%	AB	0.0000	0.00%
A$\overline{\text{B}}$C	0.9999	+0%	+0%	AC	0.0000	3.70%
AB$\overline{\text{C}}$	0.9998	+0%	+0%	BC	0.0000	0.00%
ABC	0.9999	+0%	+0%	ABC	0.0000	0.00%

Correlation with `lisp` performance: $\rho = +0.994$

Table 5.25: Instruction cache hits, `troff`

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	721	+0%		μ	1543	
$\overline{\text{AB}}$C	720	-0%		A	823	95.86%
$\overline{\text{A}}$B$\overline{\text{C}}$	721	+0%		B	-21	0.06%
$\overline{\text{A}}$BC	720	-0%		C	119	2.00%
A$\overline{\text{BC}}$	2158	+199%	+3%	AB	-21	0.06%
A$\overline{\text{B}}$C	2658	+269%	+27%	AC	119	2.00%
AB$\overline{\text{C}}$	2098	+191%	+0%	BC	-6	0.01%
ABC	2550	+254%	+22%	ABC	-6	0.01%

Correlation with `troff` performance: $\rho = +0.966$

Table 5.26: Instruction cache fetches, `troff`

Trial	fetches ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	769	+0%		μ	1603	
$\overline{\text{AB}}$C	769	+0%		A	834	95.96%
$\overline{\text{A}}$B$\overline{\text{C}}$	769	+0%		B	-21	0.06%
$\overline{\text{A}}$BC	769	+0%		C	119	1.95%
A$\overline{\text{BC}}$	2229	+190%	+3%	AB	-21	0.06%
A$\overline{\text{B}}$C	2729	+255%	+26%	AC	119	1.95%
AB$\overline{\text{C}}$	2169	+182%	+0%	BC	-6	0.00%
ABC	2621	+241%	+21%	ABC	-6	0.00%

Correlation with `troff` performance: $\rho = +0.966$

Table 5.27: Instruction cache hit rate, `troff`

Trial	hit rate	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.9368	+0%		μ	0.9536	
$\overline{\text{AB}}$C	0.9368	+0%		A	0.0168	98.49%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.9368	+0%		B	-0.0003	0.02%
$\overline{\text{A}}$BC	0.9368	+0%		C	0.0015	0.73%
A$\overline{\text{BC}}$	0.9680	+3%	+0%	AB	-0.0003	0.02%
A$\overline{\text{B}}$C	0.9739	+4%	+1%	AC	0.0015	0.73%
AB$\overline{\text{C}}$	0.9671	+3%	+0%	BC	-0.0000	0.00%
ABC	0.9728	+4%	+1%	ABC	-0.0000	0.00%

Correlation with `troff` performance: $\rho = +0.945$

Table 5.28: Instruction cache hits, all experiments

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	1411	+0%		μ	2879	
$\overline{\text{AB}}$C	1411	+0%		A	1468	95.97%
$\overline{\text{A}}$B$\overline{\text{C}}$	1411	+0%		B	-33	0.05%
$\overline{\text{A}}$BC	1411	+0%		C	210	1.96%
A$\overline{\text{BC}}$	3974	+182%	+2%	AB	-33	0.05%
A$\overline{\text{B}}$C	4851	+244%	+25%	AC	210	1.96%
AB$\overline{\text{C}}$	3879	+175%	+0%	BC	-9	0.00%
ABC	4683	+232%	+21%	ABC	-9	0.00%

Correlation with overall performance: $\rho = +0.976$

Table 5.29: Instruction cache fetches, all experiments

Trial	fetches ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	1460	+0%		μ	2939	
$\overline{\text{AB}}$C	1460	+0%		A	1479	96.03%
$\overline{\text{A}}$B$\overline{\text{C}}$	1460	+0%		B	-33	0.05%
$\overline{\text{A}}$BC	1460	+0%		C	210	1.94%
A$\overline{\text{BC}}$	4045	+177%	+2%	AB	-33	0.05%
A$\overline{\text{B}}$C	4922	+237%	+25%	AC	210	1.94%
AB$\overline{\text{C}}$	3951	+171%	+0%	BC	-9	0.00%
ABC	4755	+226%	+20%	ABC	-9	0.00%

Correlation with overall performance: $\rho = +0.975$

Table 5.30: Instruction cache hit rate, all experiments

Trial	hit rate	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.9666	+0%		μ	0.9751	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	0.9666	+0%		A	0.0085	98.33%
$\overline{\text{A}}\text{B}\overline{\text{C}}$	0.9666	+0%		B	-0.0001	0.02%
$\overline{\text{A}}\text{BC}$	0.9666	+0%		C	0.0008	0.81%
A$\overline{\text{BC}}$	0.9823	+2%	+0%	AB	-0.0001	0.02%
A$\overline{\text{B}}$C	0.9855	+2%	+0%	AC	0.0008	0.81%
AB$\overline{\text{C}}$	0.9819	+2%	+0%	BC	-0.0000	0.00%
ABC	0.9849	+2%	+0%	ABC	-0.0000	0.00%

Correlation with overall performance: $\rho = +0.959$

The statistics on data cache behavior are quite interesting. Data cache rates are expected to drop somewhat in a copying garbage collector, since moving data requires invalidating any cache copies of it that exist. The cache rates reported here, however, drop much more extremely than one would expect. The variation in data cache rates shows a moderately high degree of correlation with overall performance, although not so high as that exhibited by a number of other parameters; this lesser correlation is probably due to the *expected* drop in cache hit rates being combined with the *unexpected* drop in cache hit rates attributable to stack manipulation overhead (discussed below).

Table 5.31: Data cache hits, `sfft`

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	107	+0%		μ	105	
$\overline{\text{AB}}$C	107	+0%		A	-3	100.00%
$\overline{\text{AB}}\overline{\text{C}}$	107	+0%		B	0	0.00%
$\overline{\text{A}}$BC	107	+0%		C	0	0.00%
A$\overline{\text{BC}}$	102	-5%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	102	-5%	+0%	AC	0	0.00%
AB$\overline{\text{C}}$	102	-5%	+0%	BC	0	0.00%
ABC	102	-5%	+0%	ABC	0	0.00%

Correlation with `sfft` performance: ρ = -0.842

The statistics on data cache hits and data cache fetches show that the variation in hit rates has two separate components. In all cases, the number of data hits goes down slightly when switching to the garbage-collection architecture, while the number of data fetches increases by a much larger amount. The drop in the number of cache hits probably reflects the normal expected drop due to the use of a copying collector. The more significant increase in the number of fetches can likely be attributed to the busy-waiting that takes place while waiting for stack manipulation operations to complete. While waiting, the mutator spins in a four-instruction loop that continually reads from the `GCResult` register of the memory arbiter. Since this is a fetch of a memory operand, and since the `GCResult` register is volatile and therefore uncachable, each

Table 5.32: Data cache fetches, `sfft`

Trial	fetches ($\times 10^6$)	:$\overline{\text{A}}$BC	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{A}}\overline{\text{B}}\overline{\text{C}}$	108	+0%		μ	114	
$\overline{\text{A}}\overline{\text{B}}$C	108	+0%		A	6	94.74%
$\overline{\text{A}}$B$\overline{\text{C}}$	108	+0%		B	0	0.00%
$\overline{\text{A}}$BC	108	+0%		C	1	2.63%
A$\overline{\text{B}}\overline{\text{C}}$	117	+8%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	122	+13%	+4%	AC	1	2.63%
AB$\overline{\text{C}}$	117	+8%	+0%	BC	0	0.00%
ABC	122	+13%	+4%	ABC	0	0.00%

Correlation with `sfft` performance: $\rho = +0.943$

Table 5.33: Data cache hit rate, `sfft`

Trial	hit rate	:$\overline{\text{A}}$BC	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{A}}\overline{\text{B}}\overline{\text{C}}$	0.9969	+0%		μ	0.9233	
$\overline{\text{A}}\overline{\text{B}}$C	0.9969	+0%		A	-0.0736	97.80%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.9969	+0%		B	0.0000	0.00%
$\overline{\text{A}}$BC	0.9969	+0%		C	-0.0078	1.10%
A$\overline{\text{B}}\overline{\text{C}}$	0.8653	-13%	+0%	AB	0.0000	0.00%
A$\overline{\text{B}}$C	0.8341	-16%	-4%	AC	-0.0078	1.10%
AB$\overline{\text{C}}$	0.8653	-13%	+0%	BC	0.0000	0.00%
ABC	0.8341	-16%	-4%	ABC	0.0000	0.00%

Correlation with `sfft` performance: $\rho = -0.903$

Table 5.34: Data cache hits, `lisp`

Trial	hits ($\times 10^5$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	769	+0%		μ	662	
$\overline{\text{AB}}$C	769	+0%		A	-107	100.00%
$\overline{\text{A}}$B$\overline{\text{C}}$	769	+0%		B	0	0.00%
$\overline{\text{A}}$BC	769	+0%		C	0	0.00%
A$\overline{\text{BC}}$	555	-28%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	555	-28%	+0%	AC	0	0.00%
AB$\overline{\text{C}}$	555	-28%	+0%	BC	0	0.00%
ABC	555	-28%	+0%	ABC	0	0.00%

Correlation with `lisp` performance: ρ = -0.943

Table 5.35: Data cache fetches, `lisp`

Trial	fetches ($\times 10^5$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	774	+0%		μ	1627	
$\overline{\text{AB}}$C	774	+0%		A	853	92.34%
$\overline{\text{A}}$B$\overline{\text{C}}$	774	+0%		B	-24	0.07%
$\overline{\text{A}}$BC	774	+0%		C	172	3.75%
A$\overline{\text{BC}}$	2171	+180%	+3%	AB	-24	0.07%
A$\overline{\text{B}}$C	2882	+272%	+37%	AC	172	3.75%
AB$\overline{\text{C}}$	2101	+171%	+0%	BC	-6	0.00%
ABC	2763	+257%	+32%	ABC	-6	0.00%

Correlation with `sfft` performance: ρ = +0.938

Table 5.36: Data cache hit rate, `lisp`

Trial	hit rate	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.9938	+0%		μ	0.6110	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	0.9938	+0%		A	-0.3828	99.66%
$\overline{\text{A}}\text{BC}$	0.9938	+0%		B	0.0021	0.00%
$\overline{\text{A}}\text{BC}$	0.9938	+0%		C	-0.0158	0.17%
A$\overline{\text{BC}}$	0.2556	-74%	-3%	AB	0.0021	0.00%
A$\overline{\text{B}}$C	0.1926	-81%	-27%	AC	-0.0158	0.17%
AB$\overline{\text{C}}$	0.2641	-73%	+0%	BC	-0.0000	0.00%
ABC	0.2009	-80%	-24%	ABC	-0.0000	0.00%

Correlation with `lisp` performance: ρ = -0.960

Table 5.37: Data cache hits, `troff`

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	174	+0%		μ	151	
$\overline{\text{A}}\text{B}\overline{\text{C}}$	174	+0%		A	-24	100.00%
$\overline{\text{A}}\text{BC}$	174	+0%		B	0	0.00%
$\overline{\text{A}}\text{BC}$	174	+0%		C	0	0.00%
A$\overline{\text{BC}}$	127	-27%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	127	-27%	+0%	AC	0	0.00%
AB$\overline{\text{C}}$	127	-27%	+0%	BC	0	0.00%
ABC	127	-27%	+0%	ABC	0	0.00%

Correlation with `troff` performance: ρ = -0.901

Table 5.38: Data cache fetches, `troff`

Trial	fetches ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	175	+0%		μ	302	
$\overline{\text{AB}}$C	175	+0%		A	127	93.15%
$\overline{\text{A}}$B$\overline{\text{C}}$	175	+0%		B	-4	0.09%
$\overline{\text{A}}$BC	175	+0%		C	24	3.33%
A$\overline{\text{BC}}$	387	+121%	+3%	AB	-4	0.09%
A$\overline{\text{B}}$C	487	+178%	+30%	AC	24	3.33%
AB$\overline{\text{C}}$	375	+114%	+0%	BC	-1	0.01%
ABC	465	+166%	+24%	ABC	-1	0.01%

Correlation with `troff` performance: $\rho = +0.977$

Table 5.39: Data cache hit rate, `troff`

Trial	hit rate	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.9944	+0%		μ	0.6474	
$\overline{\text{AB}}$C	0.9944	+0%		A	-0.3470	99.53%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.9944	+0%		B	0.0027	0.01%
$\overline{\text{A}}$BC	0.9944	+0%		C	-0.0167	0.23%
A$\overline{\text{BC}}$	0.3288	-67%	-3%	AB	0.0027	0.01%
A$\overline{\text{B}}$C	0.2612	-74%	-23%	AC	-0.0167	0.23%
AB$\overline{\text{C}}$	0.3390	-66%	+0%	BC	0.0002	0.00%
ABC	0.2729	-73%	-19%	ABC	0.0002	0.00%

Correlation with `troff` performance: $\rho = -0.927$

Table 5.40: Data cache hits, all experiments

Trial	hits ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	359	+0%		μ	322	
$\overline{\text{AB}}$C	359	+0%		A	-38	100.00%
$\overline{\text{A}}$B$\overline{\text{C}}$	359	+0%		B	0	0.00%
$\overline{\text{A}}$BC	359	+0%		C	0	0.00%
A$\overline{\text{BC}}$	284	-21%	+0%	AB	0	0.00%
A$\overline{\text{B}}$C	284	-21%	+0%	AC	0	0.00%
AB$\overline{\text{C}}$	284	-21%	+0%	BC	0	0.00%
ABC	284	-21%	+0%	ABC	0	0.00%

Correlation with overall performance: ρ = -0.918

Table 5.41: Data cache fetches, all experiments

Trial	fetches ($\times 10^6$)	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	360	+0%		μ	578	
$\overline{\text{AB}}$C	360	+0%		A	218	92.90%
$\overline{\text{A}}$B$\overline{\text{C}}$	360	+0%		B	-7	0.10%
$\overline{\text{A}}$BC	360	+0%		C	42	3.45%
A$\overline{\text{BC}}$	721	+100%	+3%	AB	-7	0.10%
A$\overline{\text{B}}$C	897	+149%	+28%	AC	42	3.45%
AB$\overline{\text{C}}$	703	+95%	+0%	BC	-2	0.01%
ABC	863	+140%	+23%	ABC	-2	0.01%

Correlation with overall performance: ρ = +0.986

Table 5.42: Data cache hit rate, all experiments

Trial	hit rate	:$\overline{A}BC$	:$AB\overline{C}$	Factor	Effect	% Var
$\overline{A}\overline{B}\overline{C}$	0.9950	+0%		μ	0.6781	
$\overline{A}\overline{B}C$	0.9950	+0%		A	-0.3169	99.26%
$\overline{A}B\overline{C}$	0.9950	+0%		B	0.0028	0.01%
$\overline{A}BC$	0.9950	+0%		C	-0.0191	0.36%
$A\overline{B}\overline{C}$	0.3941	-60%	-3%	AB	0.0028	0.01%
$A\overline{B}C$	0.3169	-68%	-22%	AC	-0.0191	0.36%
$AB\overline{C}$	0.4046	-59%	+0%	BC	0.0002	0.00%
ABC	0.3290	-67%	-19%	ABC	0.0002	0.00%

Correlation with overall performance: ρ = -0.947

iteration of the loop results in an additional data cache miss. The hypothesis that stack manipulation overhead is responsible for the increase in fetches is supported by the high correlation of the number of cache fetches with the number of elapsed CPU cycles, which is itself highly correlated with the number of instructions executed.

It should also be noted that increasing the CPU speed has no effect on data cache hit rates when the traditional architecture is used, but tends to exacerbate the poor hit rates of the garbage-collection architecture. The tables above show that this is entirely due to the increased number of fetches, since the number of hits remains constant. Again, this appears to be an effect of the stack manipulation problem; as the CPU speed increases, the number of iterations of the busy-waiting loop increases correspondingly.

5.4.6 Difference in costs and latencies

This statistic applies only to the garbage-collection architecture, so only those trials are listed here. The statistic measures the total number of cycles over all test cases that are wasted because the mutator does not claim the results from the arbiter exactly when they become ready. In the raw data in reference [44], an *average cost* and an *average latency* figure are given for each arbiter operation. The cost is the number of cycles required by the garbage-collection module to process a request, from

the time the last word of data is received from the mutator until the result (if any) is ready or the operation has otherwise completed. The latency is the number of cycles from the time the last word of data is received from the mutator until the mutator actually claims the result. The difference between average costs and average latencies is reported in Tables 5.43–5.46.

Although this statistic correlates closely with the overall performance of the different implementations of the garbage-collection architecture, this correlation does not appear to be very meaningful. The effect of the size of garbage-collected memory appears to be too small to be conclusive, while the difference between costs and latencies for the `troff` cases unexpectedly increases when a fast CPU is used. In fact, comparing this statistic over different alternatives provides little information, since the difference in costs and latencies is primarily due to how "lucky" the mutator is when it claims a result; it will have to execute between zero and four instructions after the time when the result becomes ready, and the average of this amount does not say much about the relative merits of the different configurations. The more pertinent information about this statistic is the fraction of the total elapsed CPU cycles that are wasted in this protocol. Table 5.47 shows this information for the standard garbage-collection configuration. Note that the wasted time is fairly significant in both the `lisp` and `troff` test cases. Tables 5.48–5.51 show the value of this fraction for all trials in the usual manner.

Note that in all cases except that of `sfft`, the number of cycles wasted in this protocol is a significant percentage of the total number of elapsed cycles. This is in large part due to the large number of stack manipulation operations, which must be somehow reduced; but even so, this statistic demonstrates the waste involved in forcing the mutator to perform a busy-waiting loop until the arbiter has produced its results. A better alternative would be to have the arbiter raise a stall signal on the system bus whenever the mutator tries to claim a result that is not yet ready, and lower the stall signal when the result becomes available; in this way the mutator gets the result as fast as possible, and does not flood the bus with repeated requests for the same result. Such flooding would be extremely undesirable in a multiprocessing environment. Of course, care must be taken to ensure that the bus is not stalled

Table 5.43: Total cycles difference between costs and latencies, `sfft`

Trial	cycles ($\times 10^3$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	5436	+0%	μ	6662	
A$\overline{\text{B}}$C	7888	+45%	B	0	0.00%
AB$\overline{\text{C}}$	5436	+0%	C	1226	100.00%
ABC	7888	+45%	BC	0	0.00%

Correlation with `sfft` performance: $\rho = +1.000$

Table 5.44: Total cycles difference between costs and latencies, `lisp`

Trial	cycles ($\times 10^6$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	133	+3%	μ	151	
A$\overline{\text{B}}$C	189	+47%	B	-10	17.73%
AB$\overline{\text{C}}$	129	+0%	C	20	70.92%
ABC	154	+19%	BC	-8	11.35%

Correlation with `lisp` performance: $\rho = +0.950$

Table 5.45: Total cycles difference between costs and latencies, `troff`

Trial	cycles ($\times 10^6$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	161	+8%	μ	167	
A$\overline{\text{B}}$C	173	+16%	B	0	0.00%
AB$\overline{\text{C}}$	149	+0%	C	12	80.00%
ABC	184	+23%	BC	6	20.00%

Correlation with `troff` performance: $\rho = +0.805$

Table 5.46: Total cycles difference between costs and latencies, all experiments

Trial	cycles ($\times 10^6$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	300	+6%	μ	325	
A$\overline{\text{B}}$C	370	+30%	B	-10	8.38%
AB$\overline{\text{C}}$	284	+0%	C	33	91.28%
ABC	346	+22%	BC	-2	0.34%

Correlation with overall performance: $\rho = +0.999$

Table 5.47: Wasted cycles for the standard GC configuration

Program	Wasted cycles	Elapsed cycles	Percent wasted
`sfft`	5,436	717,606	0.76%
`lisp`	129,298	2,766,890	4.67%
`troff`	149,342	4,880,618	3.06%
Total	284,076	8,365,114	3.40%

Table 5.48: Fraction of time wasted by protocol, `sfft`

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0076	+0%	μ	0.0080	
A$\overline{\text{B}}$C	0.0085	+12%	B	0.0000	0.00%
AB$\overline{\text{C}}$	0.0076	+0%	C	0.0005	100.00%
ABC	0.0085	+12%	BC	0.0000	0.00%

Correlation with `sfft` performance: $\rho = +1.000$

Table 5.49: Fraction of time wasted by protocol, `lisp`

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0443	-5%	μ	0.0425	
A$\overline{\text{B}}$C	0.0414	-11%	B	-0.0003	1.06%
AB$\overline{\text{C}}$	0.0467	+0%	C	-0.0030	78.09%
ABC	0.0376	-19%	BC	-0.0015	20.85%

Correlation with `lisp` performance: ρ = -0.792

Table 5.50: Fraction of time wasted by protocol, `troff`

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0294	-4%	μ	0.0265	
A$\overline{\text{B}}$C	0.0205	-33%	B	0.0015	15.45%
AB$\overline{\text{C}}$	0.0306	+0%	C	-0.0035	78.75%
ABC	0.0255	-17%	BC	0.0009	5.80%

Correlation with `troff` performance: ρ = -0.987

Table 5.51: Fraction of time wasted by protocol, all experiments

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0326	-4%	μ	0.0303	
A$\overline{\text{B}}$C	0.0265	-22%	B	0.0008	6.35%
AB$\overline{\text{C}}$	0.0340	+0%	C	-0.0030	93.59%
ABC	0.0282	-17%	BC	0.0001	0.06%

Correlation with overall performance: ρ = -0.997

for long periods of time. Future studies are planned that will test the architecture's performance with different protocols.

5.4.7 Allocations impeded by garbage collection

Each time the mutator allocates an object, the simulator checks the garbage collector's status. As described more fully elsewhere, each allocation requires that an amount of garbage collection proportional to the size of the new object must be performed before the new object can be returned. Since the garbage-collection microprocessor runs fully independently from the mutator processor, it is often able to keep ahead of the amount of garbage collection required by mutator allocations, with the result that the mutator does not have to wait on garbage collection when it requests a new object. The simulator counts the number of these "unimpeded allocations." It turns out that, in all of the experiments carried out here, not a single allocation was ever impeded by garbage collection. This would appear to indicate that the algorithm could be much more flexible and still be quite robust. That is, it may be unnecessary to strictly enforce the requirement of a proportional amount of garbage collection upon allocation, since the garbage-collection microprocessor tends always to be ahead of allocation rates. However, this statistic may be misleading because of the severe overhead incurred by the stack manipulation operations. It is probable that allocation rates would be much higher if this problem were resolved. Also, a multiprocessor environment sharing use of a single garbage-collecting memory module might exceed the garbage collector's allowable execution rate if this restriction were relaxed. Careful consideration is necessary before removing the pacing constraint.

5.4.8 Cycles required for garbage collection

This statistic measures the total number of CPU cycles during which the garbage-collection processor has useful work to do. The results for this statistic, reported in Tables 5.53–5.56, again emphasize the importance of having sufficient amounts of garbage-collected memory available for use by the program, in order to avoid overutilization of the heap. The combination of a very fast mutator CPU and a small garbage-collected memory has the strongest effect, as shown in the tables; in the

Table 5.52: Number of allocations impeded by GC, all experiments

Trial	allocations	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0	+0%	μ	0	
A$\overline{\text{B}}$C	0	+0%	B	0	0.00%
AB$\overline{\text{C}}$	0	+0%	C	0	0.00%
ABC	0	+0%	BC	0	0.00%

Correlation with overall performance: $\rho = +\text{NaN}$

worst case (`troff`) the microprocessor was busy almost 22 times as much for this configuration as it was in the standard garbage-collection configuration. Of course, the overall effect on performance (see section 5.4.1) is not nearly this strong. It is important to try to keep the amount of garbage-collection activity low if possible, however, since an active garbage collector contends with the mutator for access to the heap. Garbage collection also produces additional cache invalidation requests, which increase bus utilization both directly and by lowering the data cache hit rate.

A CPU that is much faster than the garbage-collection microprocessor is also a serious source of concern. As the total number of elapsed cycles grows 46% over all test cases when varying the CPU speed (see Table 5.4), the number of cycles of garbage collection activity grows by 173%. Thus it is desirable to ensure that the microprocessor used in the garbage-collected memory module is not greatly outperformed by the mutator CPU.

5.4.9 Fraction of time that garbage collection is active

Tables 5.57–5.60 express the previous statistic as a fraction of the total number of elapsed cycles. Again it is clear that the size of garbage-collected memory and the speed of the mutator CPU have a substantial effect on garbage collection activity; but this new view of garbage collection activity puts these effects in perspective. In the worst case (running `troff` with a minimal garbage-collected memory and a fast CPU), garbage collection is still active less than 2% of the time. Again, however, this statistic should be treated cautiously. If the adverse effects of stack manipulation

Table 5.53: Total cycles required for GC, `sfft`

Trial	cycles	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0	+0%	μ	0	
A$\overline{\text{B}}$C	0	+0%	B	0	0.00%
AB$\overline{\text{C}}$	0	+0%	C	0	0.00%
ABC	0	+0%	BC	0	0.00%

Correlation with `sfft` performance: $\rho = +\text{NaN}$

Table 5.54: Total cycles required for GC, `lisp`

Trial	cycles ($\times 10^3$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	10419	+295%	μ	12166	
A$\overline{\text{B}}$C	28729	+988%	B	-7408	55.42%
AB$\overline{\text{C}}$	2640	+0%	C	5636	32.08%
ABC	6875	+160%	BC	-3519	12.51%

Correlation with `lisp` performance: $\rho = +0.758$

Table 5.55: Total cycles required for GC, `troff`

Trial	cycles ($\times 10^3$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	58048	+671%	μ	62915	
A$\overline{\text{B}}$C	165202	+2095%	B	-48710	61.95%
AB$\overline{\text{C}}$	7525	+0%	C	30128	23.70%
ABC	20884	+178%	BC	-23449	14.36%

Correlation with `troff` performance: $\rho = +0.750$

Table 5.56: Total cycles required for GC, all experiments

Trial	cycles ($\times 10^3$)	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	68467	+574%	μ	75081	
A$\overline{\text{B}}$C	193931	+1808%	B	-56119	61.08%
AB$\overline{\text{C}}$	10165	+0%	C	35765	24.81%
ABC	27759	+173%	BC	-26968	14.11%

Correlation with overall performance: $\rho = +0.730$

Table 5.57: Fraction of time GC is active, `sfft`

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0000	+0%	μ	0.0000	
A$\overline{\text{B}}$C	0.0000	+0%	B	0.0000	0.00%
AB$\overline{\text{C}}$	0.0000	+0%	C	0.0000	0.00%
ABC	0.0000	+0%	BC	0.0000	0.00%

Correlation with `sfft` performance: $\rho = +\text{NaN}$

Table 5.58: Fraction of time GC is active, `lisp`

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0035	+250%	μ	0.0031	
A$\overline{\text{B}}$C	0.0063	+530%	B	-0.0018	75.16%
AB$\overline{\text{C}}$	0.0010	+0%	C	0.0009	18.26%
ABC	0.0017	+70%	BC	-0.0005	6.58%

Correlation with `lisp` performance: $\rho = +0.646$

Table 5.59: Fraction of time GC is active, `troff`

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0106	+607%	μ	0.0086	
A$\overline{\text{B}}$C	0.0196	+1207%	B	-0.0064	80.05%
AB$\overline{\text{C}}$	0.0015	+0%	C	0.0026	13.01%
ABC	0.0029	+93%	BC	-0.0019	6.95%

Correlation with `troff` performance: $\rho = +0.654$

Table 5.60: Fraction of time GC is active, all experiments

Trial	fraction	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0074	+517%	μ	0.0062	
A$\overline{\text{B}}$C	0.0139	+1058%	B	-0.0044	78.47%
AB$\overline{\text{C}}$	0.0012	+0%	C	0.0019	14.31%
ABC	0.0023	+92%	BC	-0.0013	7.22%

Correlation with overall performance: $\rho = +0.635$

overhead can be controlled, the allocation rates will be expected to increase, driving the relative amount of garbage collection activity higher.

5.4.10 Bus utilization

Bus utilization measures the fraction of the time that the bus is active with a read, write, or invalidate request. Tables 5.61–5.64 show that CPU speed is a much more important factor in determining bus utilization than is the choice of architecture. In fact, the effect of varying the architecture is wildly different across different workloads. This statistic provides little useful information, aside from the obvious fact that a faster CPU puts greater pressure on the memory subsystems.

Table 5.61: Bus utilization, `sfft`

Trial	util.	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.277	+0%		μ	0.394	
$\overline{\text{A}}$BC	0.442	+60%		A	0.034	19.93%
$\overline{\text{A}}\overline{\text{BC}}$	0.277	+0%		B	0.000	0.00%
$\overline{\text{A}}$BC	0.442	+60%		C	0.067	76.30%
A$\overline{\text{BC}}$	0.376	+36%	+0%	AB	0.000	0.00%
A$\overline{\text{B}}$C	0.481	+74%	+28%	AC	-0.015	3.77%
AB$\overline{\text{C}}$	0.376	+36%	+0%	BC	0.000	0.00%
ABC	0.481	+74%	+28%	ABC	0.000	0.00%

Correlation with `sfft` performance: $\rho = +0.801$

Table 5.62: Bus utilization, `lisp`

Trial	util.	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.335	+0%		μ	0.404	
$\overline{\text{A}}$BC	0.495	+48%		A	-0.011	2.73%
$\overline{\text{A}}\overline{\text{BC}}$	0.335	+0%		B	-0.009	1.91%
$\overline{\text{A}}$BC	0.495	+48%		C	0.058	81.84%
A$\overline{\text{BC}}$	0.374	+12%	+10%	AB	-0.009	1.91%
A$\overline{\text{B}}$C	0.449	+34%	+32%	AC	-0.022	11.59%
AB$\overline{\text{C}}$	0.341	+2%	+0%	BC	-0.001	0.01%
ABC	0.411	+23%	+21%	ABC	-0.001	0.01%

Correlation with `lisp` performance: $\rho = +0.023$

Table 5.63: Bus utilization, `troff`

Trial	util.	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.506	+0%		μ	0.525	
$\overline{\text{A}}$BC	0.670	+32%		A	-0.063	45.70%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.506	+0%		B	-0.015	2.56%
$\overline{\text{A}}$BC	0.670	+32%		C	0.062	44.62%
A$\overline{\text{BC}}$	0.449	-11%	+15%	AB	-0.015	2.56%
A$\overline{\text{B}}$C	0.535	+6%	+37%	AC	-0.020	4.57%
AB$\overline{\text{C}}$	0.391	-23%	+0%	BC	-0.000	0.00%
ABC	0.474	-6%	+21%	ABC	-0.000	0.00%

Correlation with `troff` performance: ρ = -0.378

Table 5.64: Bus utilization, all experiments

Trial	util.	:$\overline{\text{ABC}}$	:AB$\overline{\text{C}}$	Factor	Effect	% Var
$\overline{\text{ABC}}$	0.373	+0%		μ	0.441	
$\overline{\text{A}}$B$\overline{\text{C}}$	0.536	+44%		A	-0.013	3.85%
$\overline{\text{A}}$B$\overline{\text{C}}$	0.373	+0%		B	-0.008	1.40%
$\overline{\text{A}}$BC	0.535	+43%		C	0.062	85.71%
A$\overline{\text{BC}}$	0.399	+7%	+8%	AB	-0.008	1.32%
A$\overline{\text{B}}$C	0.488	+31%	+32%	AC	-0.019	7.71%
AB$\overline{\text{C}}$	0.369	-1%	+0%	BC	-0.000	0.01%
ABC	0.455	+22%	+23%	ABC	-0.000	0.00%

Correlation with overall performance: ρ = +0.088

5.4.11 Bus utilization due to cache invalidation

The arbiter places cache invalidation requests on the system bus whenever the garbage-collection microprocessor or the arbiter overwrites data that may reside in the mutator's data cache. Tables 5.65–5.68 show the fraction of the time that the bus is active with an invalidation request. Note that almost all of the variation is explained by the speed of the CPU: as the CPU speed increases, garbage collection is active more of the time (see section 5.4.9), and hence more cache invalidation requests occur. It is interesting to note that utilization due to cache invalidation actually drops slightly when a smaller garbage-collected memory size is used. This might seem counterintuitive, since one would expect a smaller memory size to result in more frequent garbage collection, and hence in more cache invalidations. However, since the size of memory is halved, there are fewer memory words that can be cached. Thus fewer invalidations are needed during garbage collection.

Table 5.65: Bus utilization due to cache invalidation, `sfft`

Trial	utilization	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0157	+0%	μ	0.0260	
A$\overline{\text{B}}$C	0.0363	+131%	B	0.0000	0.00%
AB$\overline{\text{C}}$	0.0157	+0%	C	0.0103	100.00%
ABC	0.0363	+131%	BC	0.0000	0.00%

Correlation with `sfft` performance: $\rho = +1.000$

5.4.12 Additional statistics

A number of other statistics have been gathered, but are not sufficiently interesting to merit tabular exposition of their results. The number of cycles the mutator CPU is stalled while waiting for an instruction to be fetched, the number of cycles the CPU is stalled while waiting for memory operations to complete, and the number of cycles the CPU is stalled while waiting for a loaded operand to become available,

Table 5.66: Bus utilization due to cache invalidation, `lisp`

Trial	utilization	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0194	-8%	μ	0.0303	
A$\overline{\text{B}}$C	0.0382	+82%	B	0.0015	2.09%
AB$\overline{\text{C}}$	0.0210	+0%	C	0.0101	97.47%
ABC	0.0425	+102%	BC	0.0007	0.44%

Correlation with `lisp` performance: $\rho = +0.913$

Table 5.67: Bus utilization due to cache invalidation, `troff`

Trial	utilization	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0304	-13%	μ	0.0489	
A$\overline{\text{B}}$C	0.0593	+70%	B	0.0040	5.74%
AB$\overline{\text{C}}$	0.0348	+0%	C	0.0163	93.06%
ABC	0.0711	+104%	BC	0.0019	1.20%

Correlation with `troff` performance: $\rho = +0.820$

Table 5.68: Bus utilization due to cache invalidation, all experiments

Trial	utilization	:AB$\overline{\text{C}}$	Factor	Effect	% Var
A$\overline{\text{B}}\overline{\text{C}}$	0.0218	-8%	μ	0.0350	
A$\overline{\text{B}}$C	0.0446	+87%	B	0.0019	2.22%
AB$\overline{\text{C}}$	0.0238	+0%	C	0.0123	97.31%
ABC	0.0500	+110%	BC	0.0009	0.47%

Correlation with overall performance: $\rho = +0.894$

all are highly correlated with the overall performance.[4] This comes as no surprise, since one would expect these statistics to increase proportionally to the number of instructions executed, which has been shown to be primarily responsible for the variation in performance. The number of stalls while waiting for floating-point results to become available is uniform over all test cases. Again this is not startling; the code generated by the different compilers is identical for floating-point arithmetic.

In addition to the analysis presented here, more information can be gathered from the raw data in reference [44]. Detailed knowledge of the number, costs, and latencies of the different arbiter primitives is available therein, but is not of sufficient general interest to treat thoroughly here.

5.5 Additional experiments

In addition to the primary experiments discussed in the foregoing section, a few miscellaneous experiments were conducted for specific purposes. Section 5.5.1 investigates reducing the amount of cache invalidation required by the garbage collector, and section 5.5.2 more closely analyzes the effect of the size of garbage-collected memory on performance. The raw test results from these experiments are contained in reference [44].

5.5.1 Partial cache invalidation

It is shown in section 5.4.11 that cache invalidation requests represent a significant component of the total bus utilization. In the experiments in the previous section, the simulator was designed to invalidate the cache for all affected addresses whenever a `StackPush`, `InitBlock`, `CopyBlock`, or `CopyPush` operation takes place, and to invalidate all *from-space* addresses at the time of a flip. This was considered

[4] There is one exception to this statement. For the `troff` test cases, most of the variation in the number of stalls during instruction fetch is due to the effect of CPU speed, rather than the choice of architecture. This is because `troff`, with its larger code size, exhibits worse instruction cache behavior than the other two programs (see section 5.4.5). As CPU speed is doubled, so is the cost of a cache miss. Thus when instruction cache hit rates are low, the effect of CPU speed on instruction fetch stalls dominates the effect of architecture choice.

necessary, since the garbage collector is altering the values stored in the affected locations at these times. However, it turns out that the `StackPush` and `InitBlock` operations do not need to invalidate the cache, provided that the mutator cooperates. A well-behaved program will never read from an address that has been allocated on the stack before it has written to it, so a new activation frame does not need to be invalidated when it is created by a `StackPush`. The `InitBlock` operation is called either to initialize a newly allocated object, or to change the tag bits when writing to a union object. Again, a well-behaved program will write to a new object before reading from it. It is unnecessary to invalidate the cache when using `InitBlock` to set the tag bits for a union object, because the compiler only generates this code when the union object is about to be written to.

Since the frequency of the `StackPush` operation dominates that of the other cache-invalidating operations (see section 5.4.3), one would expect that removing cache invalidation from this operation would have a substantial effect on bus utilization. To test this hypothesis and to observe the effect of this change on the other measured statistics, the simulator was altered to not invalidate the cache during `InitBlock` and `StackPush` operations. One input set for each program was selected to be executed on the modified simulator. The factors were set to the levels $A\overline{B}C$; that is, the garbage-collection architecture is used with a small garbage-collected memory size and a normal CPU. The effect of the size of garbage-collected memory is investigated in section 5.5.2.

The results of these experiments are shown in Tables 5.69 through 5.71. Each table contains the figures for one of the experiments. Each statistic discussed in the previous section is measured using both the full and partial cache invalidation schemes. The final column in each table shows the percent increase or decrease in each statistic when full invalidation is replaced with partial invalidation.

Partial cache invalidation causes substantial improvement in the overall performance of the garbage-collection architecture. In the `lisp` and `troff` cases, which make heavy use of garbage collection, the total number of elapsed cycles is reduced by approximately 20% when `StackPush` and `InitBlock` operations do not invalidate the cache. The effect on `sfft` is lesser but still noticeable. The importance of these gains, however, must be deemphasized, since the average increase in elapsed cycles

Table 5.69: Effect of partial cache invalidation, `sfft/small`

Statistic	Full invalidation	Partial invalidation	Percent change
Elapsed cycles	145,436,075	134,837,904	-7.29%
Executed instructions	72,545,317	71,833,617	-0.99%
Allocation latencies	9,214	7,265	-21.15%
Icache hits	77,985,338	77,095,714	-1.14%
Icache fetches	77,991,967	77,102,342	-1.14%
Icache hit rate	99.992%	99.991%	-0.00%
Dcache hits	20,368,329	21,729,982	+6.69%
Dcache fetches	23,695,346	23,517,421	-0.75%
Dcache hit rate	85.959%	92.400%	+7.49%
Latencies − Costs	1,144,589	1,217,212	+6.34%
Percent waste	0.787%	0.903%	+14.74%
Cycles for GC	0	0	0%
Fraction GC active	0%	0%	0%
Bus utilization	37.497%	33.586%	-10.43%
Utilization for invalidation	1.672%	0.002%	-99.88%

when switching from the traditional architecture to the standard garbage-collection configuration is about 400% (see section 5.4.1). Although partial cache invalidation decreases the overhead of each `StackPush` operation, it cannot negate the effect of the function call overhead problem.

Partial cache invalidation also slightly reduces the number of instructions executed. This is because cache invalidation is one of the components that make up the difference between costs and latencies of arbiter operations. Therefore latencies are slightly improved for `StackPush` and `InitBlock` operations, reducing the amount of time the mutator must spend in busy waiting.

The effect on allocation latencies appears to be indeterminate, rising slightly in the case of `lisp` while falling noticeably for the `sfft` and `troff` experiments. Programs that allocate larger objects are expected to benefit from partial cache invalidation, since new objects larger than 32 words are initialized with `InitBlock` invocations. Smaller objects use `AllocInitRec` instead, which does not invalidate

Table 5.70: Effect of partial cache invalidation, `lisp/prune`

Statistic	Full invalidation	Partial invalidation	Percent change
Elapsed cycles	1,084,256,202	867,550,106	-19.99%
Executed instructions	422,534,475	420,819,029	-0.41%
Allocation latencies	867,476	876,120	+1.00%
Icache hits	507,729,196	505,584,381	-0.42%
Icache fetches	507,838,740	505,694,221	-0.42%
Icache hit rate	99.978%	99.978%	0%
Dcache hits	19,494,145	35,598,628	+82.61%
Dcache fetches	77,113,395	76,684,441	-0.56%
Dcache hit rate	25.280%	46.422%	+83.63%
Latencies − Costs	47,539,012	42,911,909	-9.73%
Percent waste	4.384%	4.946%	+12.82%
Cycles for GC	6,279,160	8,621,942	+37.31%
Fraction GC active	0.579%	0.994%	+71.68%
Bus utilization	38.006%	26.872%	-29.30%
Utilization for invalidation	1.903%	0.013%	-99.32%

the cache in either scheme, since the allocated data was invalidated at the time of a flip. Since almost all of the objects allocated in `lisp` are very small (a two-word CONS cell is typical), very few of `lisp`'s allocations benefit from partial cache invalidation. A more significant portion of the other programs' allocations are larger than 32 words, so their allocation latencies decrease accordingly. The slight increase in `lisp`'s allocation latencies is probably attributable to the indeterminate effects of the busy-waiting protocol.

Instruction cache behavior is essentially unaffected by partial cache invalidation. Both the number of fetches and the number of hits drop proportionately with the number of executed instructions, resulting in virtually unchanged hit rates. Data cache behavior, however, is much more interesting. Data cache hit rates rise dramatically with partial invalidation, with most of the increase attributable to a jump in the number of data cache hits, although the number of fetches drops slightly as well. The decrease in fetches is apparently primarily due to the decrease in the number of

Table 5.71: Effect of partial cache invalidation, `troff/osmpaper`

Statistic	Full invalidation	Partial invalidation	Percent change
Elapsed cycles	1,745,173,996	1,407,139,843	-19.37%
Executed instructions	612,749,907	501,139,688	-1.89%
Allocation latencies	2,002,472	1,866,376	-6.80%
Icache hits	703,650,950	689,131,327	-2.06%
Icache fetches	722,770,716	708,252,855	-2.01%
Icache hit rate	97.355%	97.300%	-0.06%
Dcache hits	40,709,858	65,649,965	+61.26%
Dcache fetches	124,976,974	122,071,326	-2.32%
Dcache hit rate	32.574%	53.780%	+65.10%
Latencies − Costs	52,521,728	52,179,298	-0.65%
Percent waste	3.010%	3.708%	+23.19%
Cycles for GC	11,484,009	10,843,898	-5.57%
Fraction GC active	0.658%	0.771%	+17.17%
Bus utilization	44.362%	32.323%	-27.14%
Utilization for invalidation	3.054%	0.002%	-99.93%

instructions executed. The increase in hits shows the adverse effects of unnecessarily invalidating cache lines that are about to be written to. When the CPU writes to a location that is not in the cache, it employs a *write-around* policy: the data word is updated in main memory, but is not placed in the cache. This is necessary because DLX contains byte and halfword store instructions; writing directly to the cache with these instructions would produce data inconsistent with main memory. This would also be required for a cache whose line size is greater than one word. Subsequent reads of a written-around data word incur a cache miss, causing the word to be fetched into the cache. The full invalidation scheme thus wreaks havoc with the cache performance of addresses in the activation stack, since every stack object is first invalidated, then written to, and then usually read back into the cache at a later time. The difference in data cache hits shown in the tables is due to this unnecessary bus traffic.

The difference between costs and latencies is another indeterminate statistic,

rising slightly in the `sfft` case and dropping in the others: moderately for `lisp`, marginally for `troff`. The fraction of elapsed cycles wasted by the protocol, however, increases noticeably in all cases. This is attributable to the decrease in the number of elapsed cycles.

The effect on garbage collection activity is also quite varied, rising 37% for the `lisp` experiment while dropping 6% for `troff`. The reasons for this become evident upon examination of the number of `TendingDone` operations. The `lisp` program increases from 12 flips to 16 flips when changing from full to partial cache invalidation, while the `troff` program remains steady at 2 flips. Apparently the reduced overhead of partial cache invalidation allows the mutator to create garbage at an increased rate. In the `lisp` case, using only 0x40000 bytes of garbage-collected memory, this had the effect of causing four additional flips; the larger 0x200000 bytes of memory used by `troff` was sufficient to handle the higher rate of garbage without a third flip. This suggests that, when the number of flips remains steady, the number of cycles required for garbage collection drops slightly. This probably reflects the mutator's ability to more quickly turn live data into garbage, reducing the amount of data that the garbage collector needs to copy. The fraction of time that garbage collection is active increases in both the `lisp` and `troff` cases, driven primarily by the decrease in elapsed cycles.

Most importantly, partial cache invalidation has a dramatic effect upon bus utilization, reducing it by between ten and thirty percent. This is primarily due to the elimination of unnecessary write-arounds discussed above in regards to data cache behavior. The statistics show that practically all (over 99%) of the cache invalidations are eliminated by removing them from `StackPush` and `InitBlock`. This accounts, however, for only one to three percentage points of the overall bus utilization decrease; the remaining approximately ten percent is due to the secondary effect that cache invalidation has on the activation stack.

5.5.2 The effects of garbage-collected memory size

The analysis in section 5.4 demonstrated that the size of garbage-collected memory is an important factor in determining achievable performance. Indeed, if the stack overhead problem is solved, garbage-collected memory size will likely emerge as the

Table 5.72: Effect of garbage-collected memory size

Statistic	256 KB	512 KB	1 MB	2 MB	4 MB
Elapsed cycles	3,089,144,736	2,879,081,859	2,719,844,330	2,418,967,362	2,321,634,576
Executed instructions	1,018,666,800	1,000,550,455	989,995,692	970,017,623	963,561,584
Allocation latencies	2,182,517	2,151,062	2,118,703	2,056,764	2,036,900
Icache hits	1,209,934,524	1,187,380,813	1,174,190,003	1,149,221,614	1,141,153,072
Icache fetches	1,209,996,452	1,187,397,232	1,174,206,018	1,149,237,553	1,141,168,851
Icache hit rate	99.995%	99.999%	99.999%	99.999%	99.999%
Dcache hits	50,017,795	51,703,497	51,723,355	51,723,453	51,729,354
Dcache fetches	201,780,803	197,262,425	194,623,760	189,629,248	188,015,248
Dcache hit rate	24.788%	26.211%	26.576%	27.276%	27.513%
Latencies − Costs	77,132,995	70,760,868	68,043,375	62,876,731	61,223,033
Percent waste	2.50%	2.46%	2.50%	2.60%	2.64%
Cycles for GC	2,797,451,267	9,316,934	2,870,815	1,911,786	0
Fraction GC active	90.56%	0.32%	0.11%	0.08%	0.00%
Flips	2612	8	2	1	0
Bus utilization	49.675%	47.603%	45.212%	39.826%	37.783%
Utilization for invalidation	2.915%	3.023%	3.198%	3.596%	3.746%
Invalidation cycles	90,000,000	87,000,000	87,000,000	87,000,000	87,000,000

dominant factor. This section more closely investigates the effect of memory size on the statistics analyzed in section 5.4.

These experiments concentrate on a single test case: the `lisp` program running a slightly expanded `db` input set (using two additional database queries). This test case was chosen because it executes in a very small amount of garbage-collected memory (256 kilobytes) but still generates enough garbage to require garbage collection even when using 2 megabytes of memory. The simulator was exercised on this test case in five sizes of memory ranging from 256 KB up to 4 MB. The raw test results for these experiments may be found in reference [44].

It should be noted that the `lisp` program executed here was produced with a different version of the compiler than was used for the experiments in section 5.4. By the time the present experiments were begun, preliminary results were available from the experiments in chapter 6 that investigate alternative function call mechanisms. These results showed that the SU compiler described in that chapter is superior to the other versions tested (although it does not solve the stack overhead problem). The `lisp` program used in this section was compiled using the SU compiler.

Table 5.72 shows the values of each statistic analyzed in section 5.4 for each of the

five memory sizes. The number of elapsed cycles drops by an average of 7% for each doubling of memory size, with an overall decrease of about 25% from the smallest memory to the largest. That is, the program runs about 33% faster with 4 MB of garbage-collected memory than with 256 KB. This reemphasizes the importance of maintaining sufficient garbage-collected memory for programs' needs.

The number of CPU instructions executed also shows a linear decrease as memory size increases, although the change is not as emphatic. Each doubling of memory decreases executed instructions by an average of 1.4%, with a total decrease of 5.4%. Allocation latencies show a similar decrease. These changes are explained by the fact that latencies are lower when garbage collection is idle than when it is active; garbage collection is obviously less active with larger memory sizes. The lower latencies in turn result in fewer iterations of the busy-waiting loop, explaining the lower number of CPU instructions.

Instruction cache behavior is largely unaffected by garbage-collected memory sizes, although the instruction cache hit rate drops slightly for the smallest memory size. This is probably due to the increased number of flips, which causes the code that tends the descriptors to be executed more frequently; this probably causes some other instructions to be overlaid in the cache, resulting in later cache misses. The data cache hit rate, on the other hand, increases at an average rate of 2.7% as memory sizes double, with the most important component of this average being the 5.7% increase between the two smallest memory sizes. The decreased number of data fetches, due to fewer iterations of the busy-waiting loop, is primarily responsible for this; the number of data hits is roughly steady except for the smallest memory size.

The differences between latencies and costs also decrease roughly linearly with each doubling of memory size. This is puzzling, since the same arbiter calls are made during each trial, with the exception of a slight difference in the number of `TendDesc` operations. There is no immediately apparent reason why costs and latencies should differ by less when garbage collection is idle than when it is active. Examination of the different categories of arbiter operations shows that the cause lies with the frequently-invoked `StackPush` and `StackPop` commands, but no explanatory pattern is evident. Further tests are necessary to investigate whether this observed phenomenon will recur with other workloads.

The percent of elapsed CPU cycles wasted by the difference between costs and latencies exhibits an interesting pattern: it drops slightly from the 256 KB trial to the 512 KB trial, and then rises steadily as memory sizes continue to increase. This anomaly is due to the fact that garbage collection is active much more of the time in the 256 KB test case (see below); thus the change in the differences between costs and latencies dominates the change in elapsed CPU cycles for this one case.

Up until this point, all statistics have shown roughly linear behavior as memory sizes are doubled. The number of cycles required for garbage collection, however, seems to exhibit a roughly exponential decrease as memory sizes double. Reducing memory from 512 KB to 256 KB causes a huge upswing (approximately 30,000%!) in garbage collection activity. The fraction of the time that garbage collection is active follows this parameter closely, dropping from 90.56% to 0.32% between the two smallest memory sizes, as does the number of flips. This again emphasizes the detrimental effect that an undersized garbage-collected memory can have on performance. Although the decrease in overall performance between the two smallest memory sizes does not appear that different from that exhibited by any of the other halvings of memory, more serious performance problems may be hidden by the stack manipulation overhead. When that problem is solved, the effect of the smallest memory size will likely be more pronounced.

Finally, increasing the size of garbage-collected memory causes a linear decrease in bus utilization. As the table shows, this decrease would have been greater if not for the concomitant increase in the component of bus utilization attributable to cache invalidation requests. However, the next line of the table shows that the number of cycles required for cache invalidation is roughly constant (to three significant digits) except for the smallest memory size, so this should not be interpreted as an indication that larger memory sizes require more cache invalidation. Rather, the increased ratio is due to the corresponding decrease in overall performance.

These experiments used the full cache invalidation protocol, rather than the partial invalidation protocol of the foregoing section. As shown there, over 99% of the bus utilization due to cache invalidation can be eliminated; thus the utilization due to invalidation can essentially be subtracted from the total utilization to obtain the bus utilization using partial cache invalidation. If this is done, bus utilization

drops from 46.8% for the smallest memory size to 34.0% for the largest, an overall decrease of 27%. On average, doubling memory size results in a 4.2% decrease in bus utilization.

5.6 Summary

This chapter has shown that the garbage-collection architecture, in its current configuration, does not yet perform competitively with traditional architectures. However, most of the performance loss can be attributed to a single factor: the overhead associated with the `StackPush` and `StackPop` operations used in the function call mechanism. There seem grounds for optimism that, if this overhead can be largely eliminated, the garbage-collection architecture will be able to compete effectively. One indication of this is that the garbage-collection architecture is much more efficient at allocating new objects than is the traditional method of using `malloc()` and `free()`.

Another problem of lesser concern is the mechanism used for communicating results of arbiter operations to the mutator. Currently the mutator executes a busy-waiting loop after requesting a service, repeatedly reading the `GCStatus` or `GCResult` register until the result becomes available. In at least some cases, it may be more appropriate to stall the mutator when it requests a result until the result becomes available.

Several minor modifications can be made to the garbage-collection architecture to improve performance. Removing unnecessary cache invalidations from the `StackPush` and `InitBlock` services can improve performance by up to 20%, although any implementation that solves the stack manipulation overhead problem will largely obviate the need for this. Increasing the size of garbage-collected memory is important to avoid excess garbage collection that saps overall performance when the ratio of live data to garbage-collected memory size is high. The system designer should ensure that the microprocessor in the garbage-collection architecture is as powerful as possible; if the mutator CPU is too much faster than the garbage collection microprocessor, the pressure on garbage-collected memory increases the performance differences between it and standard memory. Finally, it appears that the pacing of allocation (by

insisting upon a proportional amount of garbage collection at each allocation) may not be necessary. This conclusion must be reinvestigated, however, in any design that solves the stack manipulation overhead problem.

Investigations are currently underway to find a solution to this critical problem. One promising alternative is described in section 6.3 of the following chapter.

6. ALTERNATIVE FUNCTION CALL MECHANISMS

This chapter details the tradeoffs between a variety of mechanisms that can be used to implement C++ function calls using the garbage-collected memory module architecture. As discussed above, the garbage collector supports special stack objects that can be used to implement a function activation stack. When the hardware was initially designed, it was hypothesized that such special-purpose stack objects would provide better performance than the obvious alternative scheme in which each activation frame is separately allocated from the heap. The empirical results discussed in this chapter investigate, among other things, the validity of this hypothesis. Two of the four alternative compilers measured in this chapter use heap allocation of activation frames instead of the special stack object.

As mentioned briefly in section 4.1.4.1, the modified compiler used in the experiments of chapter 5 (hereafter called the *base* compiler) performs two `StackPush` operations per function call. The first `StackPush` pushes the caller's arguments, while the second pushes space for the callee's local variables and saved register storage. This mirrors the original GNU C++ compiler function call mechanism.

The alternative compilers measured in this chapter reduce the number of `StackPush` operations to one per function call (or, in the case of the heap-allocating compilers, zero per function call). This is done by reserving space for the caller's arguments in the caller's activation frame. There are two alternatives for reserving this space. The compiler can generate sufficient space for the *largest* argument block required for *any* function call in the body of the caller, or it can generate *separate* argument space for *each* such function call. If a single shared argument block is used, different function calls are likely to require different tag bits for their arguments; thus, it is necessary to perform an `InitBlock` operation to initialize the common space before each function call. If separate argument blocks are used for each func-

tion, the overhead of the `InitBlock` operation disappears; but the size of the caller's activation frame increases, thus increasing the cost of the `StackPush` operation for the frame. It is not clear *a priori* which of these two mechanisms provides better performance under real workloads.

It is also not clear whether the tradeoffs between these two mechanisms are independent of the tradeoffs between heap and stack allocation of activation frames. Thus this chapter compares the performance of the base compiler with the performances of four alternative compilers representing the four possibilities given by varying these two parameters. The characteristics of the compilers under study are summarized in Table 6.1. The alternative compilers are given two-character designations, where the first character specifies S or H for stack or heap allocation, and the second character specifies S or U for shared or unshared argument blocks.

Table 6.1: Characteristics of alternative compilers

Compiler	Argument allocation	Frame allocation	Argument blocks
Base	At function call	Stack	N/A
SS	In caller frame	Stack	Shared
SU	In caller frame	Stack	Unshared
HS	In caller frame	Heap	Shared
HU	In caller frame	Heap	Unshared

6.1 Implementation

Figures 6.1 through 6.3 illustrate the format of a typical activation frame for each of the five compilers. The SS and SU compilers have the same activation frame format, as do the HS and HU compilers, with the exception that the argument block for the SU and HU compilers is generally larger than that of the SS and HS compilers. Note that the four alternative compilers require additional dedicated registers to remember the locations of the arguments in function calls. In the base compiler, the caller pushes its arguments on the stack by varying the position of the stack pointer (r14). At the

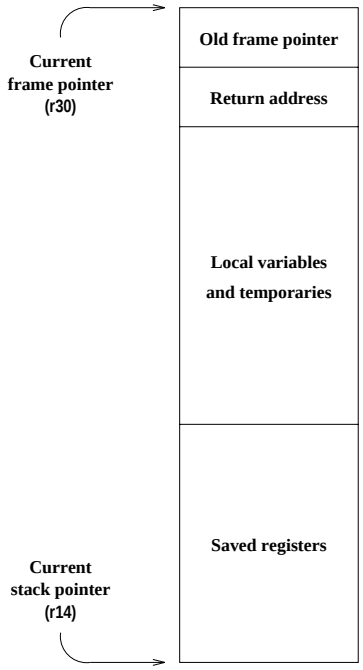

Figure 6.1: Activation frame for the base compiler

time of a function call, the new activation frame is pushed below[1] the arguments for the call, and the frame pointer is set to the value of the stack pointer prior to the push. Thus a function knows that the arguments provided by its caller are always located just above the frame pointer, and that it is expected to push arguments to functions it calls using the stack pointer. Since the alternative compilers store their arguments *within* the caller's activation frame, a separate register, the *argument pointer* (r23) is used to point to the current location of arguments to called functions. Another register (r24) contains the *parent argument pointer*, indicating where the arguments to the current function are located.

The alternative compilers do not make as good use of registers as is possible. For instance, the stack-based compilers no longer need a frame pointer, since it no

[1] In this discussion, "below" and "above" refer to relative positioning with respect to hardware memory addresses. Since the stack grows downward, the current top-of-stack is really the lowest memory address in the stack; hence a new activation frame is placed "below" its arguments.

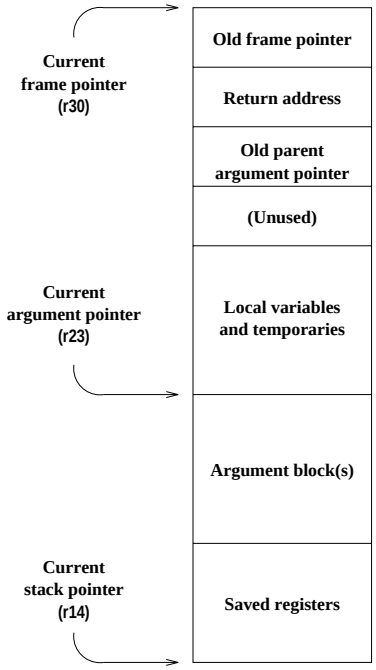

Figure 6.2: Activation frame for the SS and SU compilers

longer points to arguments and is thus redundant with the stack pointer. However, the original GNU compiler allocates all local variables and temporaries relative to the frame pointer before the total size of the activation frame is known. Although it would be relatively straightforward to modify the compiler to later alter these definitions to be relative to the stack pointer, this was not deemed worthwhile within the limited scope of this effort. Similarly, the heap-based compilers no longer have a stack, so the stack pointer is unnecessary as long as there is a frame pointer. The HS and HU compilers make no use of register `r14` as a stack pointer; but they do not reclaim it for use in normal register allocation. This is because the original compiler requires a stack pointer register to be defined; the pervasiveness of this assumption renders it virtually impossible to reuse the register as a general-purpose register. It could have, however, been reassigned for use as one of the other special-purpose registers (such as the `gcdata` base pointer, for example). Production-quality versions of any of these compilers could easily use one fewer dedicated register, slightly relieving register pressure.

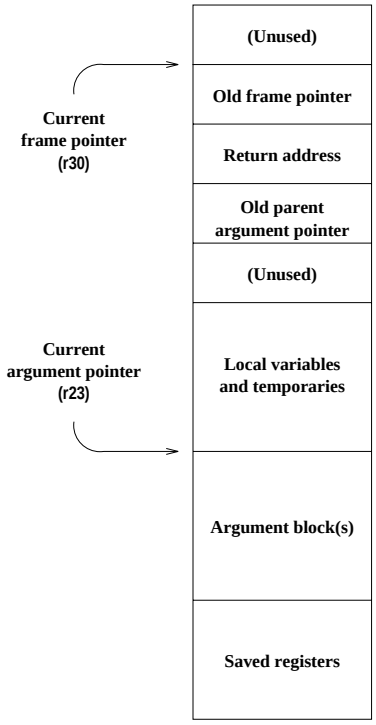

Figure 6.3: Activation frame for the HS and HU compilers

The unused word of storage below the "old parent argument pointer" is also an artifact of the original DLX compiler, which requires local variables and register-save blocks to be doubleword-aligned. This waste would also be removed in a production compiler. The other unused word depicted in Figure 6.3, on the other hand, is important to the correct operation of the garbage collector. If this word were not added, the frame pointer would actually point outside the activation frame to a neighboring object, causing the neighbor to be considered live even if it would otherwise be garbage.

Figures 6.4–6.8 illustrate example function prologue and epilogue code for each of the five compilers. The stack-based compilers begin (after saving the return address) by issuing a call to the library routine `__gc_stack_push`, which calls the arbiter's `StackPush` primitive with the arguments specified by the PLD addressed by register `r25`. The heap-based compilers call `__gc_alloc_init_rec` instead, which uses the `AllocInitRec` primitive to allocate a new record object, whose size is specified by

```
        .align 2
.global _get_sample__FP6_iobufPd
_get_sample__FP6_iobufPd:
        ;; Prologue
        ;; Save the return address
        add r27,r0,r31
        ;; Perform an IO_GC_StackPush call
        lhi   r25,(L25>>16)&0xffff
        jal   __gc_stack_push
        addui r25,r25,(L25&0xffff)      ;; (in branch delay slot)
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
        sw -8(r14),r27
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-48
        ;; Save Registers
        sw 0(r14),r15
        sw 4(r14),r16
        sd 8(r14),f4

        ;; Epilogue
        ;; Restore the saved registers
        lw r15,-48(r30)
        nop
        lw r16,-44(r30)
        nop
        ld f4,-40(r30)
        nop
        ;; Restore return address
        lw r5,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Perform an IO_GC_StackPop call
        jal   __gc_stack_pop
        addi  r3,r0,#12 ;; (in branch delay slot)
        ;; Return
        jr r5
        nop
```

Figure 6.4: Function prologue and epilogue code for base compiler

```
        .align 2
.global _get_sample__FP6_iobufPd
_get_sample__FP6_iobufPd:
        ;; Prologue
        ;; Save the return address
        add r27,r0,r31
        ;; Perform an IO_GC_StackPush call
        lhi   r25,(L40>>16)&0xffff
        jal   __gc_stack_push
        addui r25,r25,(L40&0xffff)      ;; (in branch delay slot)
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
        sw -8(r14),r27
        ;; Save the old argument pointer
        sw -12(r14),r24
        add r24,r0,r23
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-64
        ;; Establish new argument pointer
        subi r23,r30,#40
        ;; Save Registers
        sw 0(r14),r13
        sw 4(r14),r15
        sd 8(r14),f4

        ;; Epilogue
        ;; Restore the saved registers
        lw r13,-64(r30)
        nop
        lw r15,-60(r30)
        nop
        ld f4,-56(r30)
        nop
        ;; Restore argument pointer
        add r23,r0,r24
        lw r24,-12(r30)
        ;; Restore return address
        lw r5,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Perform an IO_GC_StackPop call
        jal   __gc_stack_pop
        addi  r3,r0,#16 ;; (in branch delay slot)
        ;; Return
        jr r5
        nop
```

Figure 6.5: Function prologue and epilogue code for SS compiler

```
        .align 2
.global _get_sample__FP6_iobufPd
_get_sample__FP6_iobufPd:
        ;; Prologue
        ;; Save the return address
        add r27,r0,r31
        ;; Perform an IO_GC_StackPush call
        lhi   r25,(L44>>16)&0xffff
        jal   __gc_stack_push
        addui r25,r25,(L44&0xffff)      ;; (in branch delay slot)
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
        sw -8(r14),r27
        ;; Save the old argument pointer
        sw -12(r14),r24
        add r24,r0,r23
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-72
        ;; Establish new argument pointer
        subi r23,r30,#40
        ;; Save Registers
        sw 0(r14),r13
        sd 4(r14),f4

        ;; Epilogue
        ;; Restore the saved registers
        lw r13,-72(r30)
        nop
        ld f4,-68(r30)
        nop
        ;; Restore argument pointer
        add r23,r0,r24
        lw r24,-12(r30)
        ;; Restore return address
        lw r5,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Perform an IO_GC_StackPop call
        jal   __gc_stack_pop
        addi  r3,r0,#18 ;; (in branch delay slot)
        ;; Return
        jr r5
        nop
```

Figure 6.6: Function prologue and epilogue code for SU compiler

```
        .align 2
.global _get_sample__FP6_iobufPd
_get_sample__FP6_iobufPd:
        ;; Prologue
        ;; Save the return address
        sw 0(r26),r31
        ;; Allocate the frame from the collector
        addi r4,r0,#68
        lhi r5,#0
        jal __gc_alloc_init_rec
        addui r5,r5,#40963
        ;; Save the return address
        lw r31,0(r26)
        sw 56(r25),r31
        ;; Save the old frame pointer
        sw 60(r25),r30
        ;; Save the old argument pointer
        sw 52(r25),r24
        add r24,r0,r23
        ;; Establish the new frame pointer
        addi r30,r25,#64
        ;; Establish new argument pointer
        subi r23,r30,#40
        ;; Save Registers
        sw -64(r30),r13
        sw -60(r30),r15
        sd -56(r30),f4

        ;; Epilogue
        ;; Restore the saved registers
        lw r13,-64(r30)
        nop
        lw r15,-60(r30)
        nop
        ld f4,-56(r30)
        nop
        ;; Restore argument pointer
        add r23,r0,r24
        lw r24,-12(r30)
        ;; Restore return address
        lw r5,-8(r30)
        nop
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Return
        jr r5
        nop
```

Figure 6.7: Function prologue and epilogue code for HS compiler

```
        .align 2
.global _get_sample__FP6_iobufPd
_get_sample__FP6_iobufPd:
        ;; Prologue
        ;; Save the return address
        sw 0(r26),r31
        ;; Allocate the frame from the collector
        addi r4,r0,#76
        lhi r5,#2
        jal __gc_alloc_init_rec
        addui r5,r5,#32849
        ;; Save the return address
        lw r31,0(r26)
        sw 64(r25),r31
        ;; Save the old frame pointer
        sw 68(r25),r30
        ;; Save the old argument pointer
        sw 60(r25),r24
        add r24,r0,r23
        ;; Establish the new frame pointer
        addi r30,r25,#72
        ;; Establish new argument pointer
        subi r23,r30,#40
        ;; Save Registers
        sw -72(r30),r13
        sd -68(r30),f4

        ;; Epilogue
        ;; Restore the saved registers
        lw r13,-72(r30)
        nop
        ld f4,-68(r30)
        nop
        ;; Restore argument pointer
        add r23,r0,r24
        lw r24,-12(r30)
        ;; Restore return address
        lw r5,-8(r30)
        nop
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Return
        jr r5
        nop
```

Figure 6.8: Function prologue and epilogue code for HU compiler

register r4 and whose tag bits are specified by register r5.[2] Next, all compilers save the registers used in the function call mechanism itself, i.e., the frame pointer, return address, and (for all but the base compiler) the old parent argument pointer. New values are then obtained for the frame pointer, stack pointer, argument pointer, and/or parent argument pointer, as needed by each compiler, following which registers used in the function body are saved. The function epilogue code mirrors the prologue, restoring the saved registers and return address. The stack-based compilers must issue a call to `__gc_stack_pop` to remove the activation frame from the stack; the heap-based compilers do not suffer this immediate overhead, but simply allow the activation frame to be reclaimed during the next garbage collection cycle.

6.1.1 Compiling function calls using shared argument blocks

Of the two methods of allocating space for arguments, the shared argument block alternative is by far the easier for which to generate code. The only complexities arise while processing nested and inline functions, and even these are quite minor.

Recall from section 4.1.4.1 that the base compiler generates a `StackPush` call each time it expands[3] a function call, determining the tag bits for the call by concatenating the PLDs of the arguments. The code to do this remains unchanged in the shared argument block compilers, with the exception that the resulting tag bits are used in an `InitBlock` call for the argument block, rather than in a `StackPush`. The argument pointer begins at the top of the argument block, as shown in Figures 6.2 and 6.3, and moves downward as arguments are pushed within the argument block. Nested function calls result in deeper pushing within the argument block. At the point of any function call, whether an outermost call or a nested call, the `InitBlock` is called with the current value of the argument pointer as the base address of the area to

[2]The `__gc_alloc_init_rec` routine is an optimization used for functions whose activation frames are no more than 32 words in length (thus requiring only one word of tag bits). For longer activation frames, the compiler generates separate calls to `__gc_alloc_rec` and `__gc_init_block`.

[3]The term *expand* is used throughout to mean "generate assembly code from a syntax tree representation."

be initialized, since the area immediately above the argument pointer contains the arguments for the current call.

The only additional responsibility of the shared argument block compilers is to determine the size of the argument block that must be reserved in the activation frame. This is done in the obvious way. That is, a variable `max_call_size` is initialized to zero at the time the compiler begins processing a new function definition. Whenever a function call within that definition is expanded, the compiler determines the size of the argument block needed to execute that call, taking care to account for the size required by any nested function calls within the call. If this size is larger than `max_call_size`, that variable is updated to the new value. At the end of the function definition, the compiler generates prologue and epilogue code in which `max_call_size` words are reserved for the shared argument block in the activation frame. The tag bits for the argument block are all initialized to zero; they will be set properly by the `InitBlock` for each function call.

Whenever a function is declared `inline`, assembly code for that function is not separately generated (unless the address of the function is taken), so the size of the argument block is not used to reserve space in the function's activation frame. Rather, the value of `max_call_size` for the function is stored with the syntax tree for the function definition. Whenever the inlined function is expanded within another function body, this stored value is used to determine the maximum size of the argument block needed anywhere within the expanded inline code. As with any other call, this size value is used in computing `max_call_size` for the enclosing function.

6.1.2 Compiling function calls using separate argument blocks

By contrast, generating code that creates separate argument blocks for each function call is far less natural and requires quite a bit of additional state information. For the discussion of this that follows, it is helpful to understand the compiler's function call expansion algorithm. The GNU C++ compiler generates code for a function call by performing the following steps in order:

- Reservation of space for arguments.

- Expansion into a machine address of the expression representing the function to be called.
- Expansion of each expression to be passed as an argument, in left-to-right order.
- Generation of assembly code to call the expanded function address with the expanded arguments.

The following paragraphs describe the modifications required to support the use of separate argument blocks.

The separate argument block compilers build up information about the argument blocks for each function definition in two variables, `size_of_arg_blocks` and `arg_blocks_pld`. The first of these accumulates the total amount of space reserved for argument blocks, and the second builds up a PLD describing this space. These are initialized to zero values at the beginning of the function definition. At the end of the function definition, they are incorporated into the function prologue and epilogue code.

When expanding any function call, the value of `size_of_arg_blocks` represents the total space already reserved for argument blocks due to previous function calls in the current function definition. Before expanding any of the arguments to the function calls, the compiler first determines the amount `s` of space required to hold all of the arguments, adding this value to `size_of_arg_blocks`. It then creates a PLD for an object of `s` words, with all tag bits set to zero, and prepends this new PLD to the existing `arg_blocks_pld`. (Prepending is required rather than appending, since the argument blocks grow downward but PLDs are interpreted to run from low to high addresses.) The zero-valued PLD is a "first approximation" of the tag bits for the arguments to the function call. As each argument is expanded, its PLD is found and used to overwrite the tag bits associated with that argument's location. The variable `next_bit` is used as an index into `arg_blocks_pld` to keep track of which tag bits apply to the next argument. `next_bit` is initialized to zero before any of the arguments are expanded, which causes it to point to the tag bit location for the first argument. After each argument is expanded, `next_bit` is updated to point to the tag bit location for the next argument.

It is possible that expansion of the function address will cause another function call to be expanded. For example, the function call address may be determined by calling a function that returns a function address. In this case, the argument block (and `arg_blocks_pld`) might grow before any arguments have been expanded, since the inner function call may also have arguments that must be stored in the argument block. Suppose that these arguments require an additional N words of argument block storage. Then after expanding the function address, `arg_blocks_pld` will have had an additional N tag bits prepended to it. In order to still address the tag bits for the first argument to the function, `next_bit` must be incremented by N.

When expanding an argument to a function call, the compiler first saves the value of `size_of_arg_blocks` in the variable `current_size`. As with the function address, it is possible that expanding an argument will cause the expansion of one or more nested function calls. Each nested call will again increase the value of `size_of_arg_blocks` and will prepend more tag bits onto `arg_blocks_pld`. If, following the expansion, `size_of_arg_blocks` exceeds `current_size` (say by M words), the compiler must again increment `next_bit` by M so that it addresses the tag bits for the expanded argument. The compiler then *overlays* the tag bits from the PLD of the argument type onto `arg_blocks_pld` at the offset given by `next_bit`, and updates `next_bit` to point at the tag bits for the next argument.

Since each function call within a procedure body has a separate argument block, the argument pointer (register r23) must be initialized to point to the correct argument block before the code for the function call is executed. The location of each argument block is relative to the location of the current activation frame, so this might logically be done by setting the argument pointer to the offset of the argument block relative to the frame pointer. Instead, however, the compiler selects the current argument block by "adjusting" the argument pointer, adding a constant to its current value before the function call, and returning it to its previous value afterwards. This is done because argument pointer adjustments are also used when nested function calls require pushing and popping within the argument block; using a uniform method for all argument pointer modifications facilitates a straightforward optimization discussed below.

As in the shared argument block case, argument block information for an inline function is stored with the syntax tree node associated with its definition. When a function definition is being saved for possible later inlining, the variables `size_of_arg_blocks` and `arg_blocks_pld` are stored as the syntax node attributes `DECL_CALL_SIZE` and `DECL_ARGPLD`, respectively. Each time the inline function is expanded within another function definition, `DECL_CALL_SIZE` is added to the outer function's `size_of_arg_blocks`, and `DECL_ARGPLD` is prepended to its `arg_blocks_pld`. As with normal function calls, the argument pointer is adjusted before and after the expanded inline function code.

The code generation as just described results in a proliferation of argument pointer adjustments. This is particularly noticeable when nested function calls appear. Figure 6.9 shows some example source code that exhibits this phenomenon. Figure 6.10 illustrates the assembly code generated for the `main()` function in Figure 6.9 according to the preceding description. (For brevity, the prologue and epilogue code has been deleted.) Note that there are two places where several consecutive instructions do nothing but add a constant to the contents of the argument pointer. This is the sort of inefficiency that would normally be improved upon during an optimization pass. Recall, however, from section 4.1.8 that the optimizer for GNU C++ 1.37.1 is not functional. Because it was felt that the excess argument adjustments constituted an unfair penalty against the separate argument block compilers in experimental comparisons, a simple peephole optimization has been added to detect adjacent constant adjustments to any register and collapse them into one. For fairness in comparisons, this optimization was added to all the compilers studied in this dissertation. Figure 6.11 shows the result of applying the optimization to the `main()` function of Figure 6.9.

6.2 Results of experiments

To compare the performance of the four alternative compilers with that of the base compiler, two of the programs used in chapter 5, `sfft` and `lisp`, were compiled with each of them. These two programs represent best- and worst-case performance of the base compiler, as shown in section 5.4.1. The compiled programs were then run

```
int f(int a, int b, int c)
{
  return a;
}

int g(int a)
{
  return a*2;
}

inline int h(int a)
{
  return a/2;
}

main()
{
  int i,j,k;

  i = f(i,g(j),h(k));
  k = g(j);
  i = h(k);
}
```

Figure 6.9: Example source causing excess argument pointer adjustments

on the simulator with the two input sets used in the chapter 5 experiments. Table 6.2 contains the resulting data for the combined `sfft` workload; Table 6.3 contains the data for `lisp`; and Table 6.4 contains the data totalled over all experiments. In all tables, the value of each statistic is shown in Roman type. The percentage difference between the base compiler and the alternate compilers is shown beneath each data value in a slanted font.

In terms of both elapsed CPU cycles and the number of instructions executed, it is clear that the SU compiler exhibits the best performance. In the case of `sfft`, whose performance was not as greatly affected by the choice of architecture, the SU

```
        sub r23,r23,#16
        lw r6,-20(r30)
        sw 0(r23),r6
        sub r23,r23,#8
        lw r6,-28(r30)
        sw 0(r23),r6
        jal _g__Fi
        nop
        add r23,r23,#8
        add r3,r0,r1
        sw 4(r23),r3
        lw r3,-36(r30)
        add r4,r0,r3
        addi r5,r0,#0
        sge     r1,r4,r5
        bnez    r1,L9
        nop
        add r4,r4,#1
L9:
        sra r4,r4,#1
        j L8
        nop
L8:
        sw 8(r23),r4
        jal _f__Fiii
        nop
        sw -20(r30),r1
        add r23,r23,#16         ;;
        add r23,r23,#-24        ;; consecutive adjustments
        sub r23,r23,#8          ;;
        lw r6,-28(r30)
        sw 0(r23),r6
        jal _g__Fi
        nop
        sw -36(r30),r1
        add r23,r23,#8          ;; consecutive adjustments
        add r23,r23,#24         ;;
        lw r3,-36(r30)
        add r4,r0,r3
        addi r5,r0,#0
        sge     r1,r4,r5
        bnez    r1,L11
        nop
        add r4,r4,#1
L11:
        sra r4,r4,#1
        j L10
        nop
L10:
        sw -20(r30),r4
        addi r1,r0,#0
```

Figure 6.10: Before peephole optimization

```
        sub r23,r23,#16
        lw r6,-20(r30)
        sw 0(r23),r6
        sub r23,r23,#8
        lw r6,-28(r30)
        sw 0(r23),r6
        jal _g__Fi
        nop
        add r23,r23,#8
        add r3,r0,r1
        sw 4(r23),r3
        lw r3,-36(r30)
        add r4,r0,r3
        addi r5,r0,#0
        sge     r1,r4,r5
        bnez    r1,L9
        nop
        add r4,r4,#1
L9:
        sra r4,r4,#1
        j L8
        nop
L8:
        sw 8(r23),r4
        jal _f__Fiii
        nop
        sw -20(r30),r1
        add r23,r23,#-16         ;; adjustments have been collapsed
        lw r6,-28(r30)
        sw 0(r23),r6
        jal _g__Fi
        nop
        sw -36(r30),r1
        add r23,r23,#32          ;; adjustments have been collapsed
        lw r3,-36(r30)
        add r4,r0,r3
        addi r5,r0,#0
                ;cmpsi  r4,r5
        sge     r1,r4,r5
        bnez    r1,L11
        nop
        add r4,r4,#1
L11:
        sra r4,r4,#1
        j L10
        nop
L10:
        sw -20(r30),r4
        addi r1,r0,#0
```

Figure 6.11: After peephole optimization

Table 6.2: Comparative compiler performance, `sfft`

Statistic	**Base**	**SS**	**SU**	**HS**	**HU**
Elapsed cycles	717,605,727	709,707,990 *-1.10%*	704,726,699 *-1.79%*	946,445,169 *+31.89%*	935,999,991 *+30.43%*
Executed instructions	358,113,973	352,909,229 *-1.45%*	349,351,140 *-2.45%*	347,003,031 *-3.10%*	341,702,403 *-4.58%*
CPI	2.004	2.011 *+0.35%*	2.017 *+0.65%*	2.727 *+36.08%*	2.739 *+36.68%*
Allocation latencies (includes `InitBlocks`)	18,428	7,507,922 *+40,642%*	18,518 *+0.49%*	58,914,027 *+319,598%*	57,504,478 *+311,949%*
Icache hits	384,384,406	377,392,213 *-1.82%*	373,370,303 *-2.87%*	368,667,535 *-4.09%*	362,425,089 *-5.71%*
Icache fetches	384,398,347	377,406,338 *-1.82%*	373,386,905 *-2.86%*	368,682,960 *-4.09%*	362,448,794 *-5.71%*
Icache hit rate	99.996%	99.996% *0%*	99.996% *0%*	99.996% *0%*	99.993% *-0.003%*
Dcache hits	101,645,219	101,640,118 *-0.005%*	101,211,916 *-0.43%*	101,159,042 *-0.48%*	100,712,826 *-0.92%*
Dcache fetches	117,472,569	117,040,925 *-0.37%*	116,834,302 *-0.54%*	116,710,061 *-0.65%*	116,067,345 *-1.20%*
Dcache hit rate	86.527%	86.842% *+0.36%*	86.629% *+0.12%*	86.676% *+0.17%*	86.771% *+0.28%*
Arbiter operations	1,504,541	1,165,328 *-22.55%*	908,254 *-39.63%*	911,041 *-39.45%*	654,047 *-56.53%*
Latencies — Costs	5,436,401	4,818,801 *-11.36%*	3,430,897 *-36.89%*	4,288,157 *-21.12%*	3,571,097 *-34.31%*
Percent waste	0.758%	0.679% *-10.42%*	0.487% *-35.75%*	0.453% *-40.24%*	0.382% *-49.60%*
Cycles for GC	0	0 *0%*	0 *0%*	59,563,341 *+∞%*	69,060,370 *+∞%*
Fraction GC active	0.0%	0.0% *0%*	0.0% *0%*	6.29% *+∞%*	7.38% *+∞%*
Flips	0	0 *0%*	0 *0%*	109 *+∞%*	126 *+∞%*
Total allocations	6	6 *0%*	6 *0%*	339,100 *+5,651,567%*	339,117 *+5,651,850%*
Impeded allocations	0	0 *0%*	0 *0%*	252 *+∞%*	374 *+∞%*
Percent impeded	0%	0% *0%*	0% *0%*	0.074% *+∞%*	0.110% *+∞%*
Bus utilization	37.578%	38.357% *+2.07%*	38.714% *+3.02%*	54.594% *+45.28%*	54.941% *+46.21%*
Utilization for invalidation	1.574%	1.852% *+17.66%*	2.003% *+27.26%*	0.931% *-40.85%*	0.989% *-37.17%*

Table 6.3: Comparative compiler performance, `lisp`

Statistic	**Base**	**SS**	**SU**	**HS**	**HU**
Elapsed cycles	2,766,890,083	2,668,283,456 *-3.56%*	2,254,043,552 *-18.54%*	2,847,836,033 *+2.93%*	2,599,367,225 *-6.05%*
Executed instructions	1,163,656,884	1,026,676,142 *-11.77%*	852,095,631 *-26.77%*	976,860,341 *-16.05%*	901,474,159 *-22.53%*
CPI	2.378	2.599 *+9.294%*	2.645 *+11.228%*	2.915 *+22.582%*	2.883 *+21.236%*
Allocation latencies (includes `InitBlocks`)	1,973,969	152,263,497 *+7,613%*	2,000,862 *+1.36%*	813,451,948 *+41,108%*	890,150,817 *+44.904%*
Icache hits	1,396,900,851	1,215,949,609 *-12.95%*	1,010,086,043 *-27.69%*	1,117,387,270 *-20.01%*	1,020,169,953 *-26.97%*
Icache fetches	1,397,152,600	1,216,153,201 *-12.95%*	1,010,111,155 *-27.70%*	1,117,707,877 *-20.00%*	1,021,704,853 *-26.87%*
Icache hit rate	99.982%	99.983% *-0.001%*	99.998% *+0.02%*	99.971% *-0.01%*	99.850% *-0.13%*
Dcache hits	55,504,651	55,502,684 *-0.004%*	44,615,511 *-19.62%*	46,355,319 *-16.48%*	35,913,311 *-35.30%*
Dcache fetches	210,146,023	194,123,091 *-7.62%*	167,105,764 *-20.48%*	208,742,773 *-0.67%*	206,007,810 *-1.97%*
Dcache hit rate	26.412%	28.591% *+8.25%*	26.699% *+1.09%*	22.207% *-15.92%*	17.433% *-1.97%*
Arbiter operations	29,006,846	21,773,202 *-24.94%*	14,710,168 *-49.29%*	14,556,129 *-49.82%*	7,668,846 *-73.56%*
Latencies — Costs	129,298,446	168,564,448 *+30.37%*	57,323,932 *-55.67%*	105,309,381 *-18.55%*	57,965,469 *-55.17%*
Percent waste	4.673%	6.317% *+35.18%*	2.543% *-45.58%*	3.698% *-20.86%*	2.230% *-52.28%*
Cycles for GC	2,640,265	2,723,808 *+3.16%*	3,069,843 *+16.27%*	724,139,051 *+27,326%*	871,457,591 *+32,906%*
Fraction GC active	0.095%	0.102% *+7.37%*	0.136% *+43.16%*	25.428% *+26,666%*	33.526% *+35,190%*
Flips	3	3 *0%*	3 *0%*	1055 *+35,066%*	1255 *+41,733%*
Total allocations	47,735	47,735 *0%*	47,735 *0%*	7,283,486 *+15,158%*	7,283,686 *+15,159%*
Impeded allocations	0	0 *0%*	0 *0%*	125,031 *+∞%*	679,675 *+∞%*
Percent impeded	0%	0% *0%*	0% *0%*	1.717% *+∞%*	9.331% *+∞%*
Bus utilization	34.102%	40.077% *+17.52%*	43.476% *+27.49%*	43.306% *+26.99%*	42.825% *+25.58%*
Utilization for invalidation	2.097%	2.876% *+37.15%*	3.332% *+58.89%*	0.864% *-58.80%*	0.708% *-66.24%*

Table 6.4: Comparative compiler performance, combined workload

Statistic	**Base**	**SS**	**SU**	**HS**	**HU**
Elapsed cycles	3,484,495,810	3,377,991,446 *-3.06%*	2,958,770,251 *-15.09%*	3,794,281,202 *+8.89%*	3,535,367,216 *+1.46%*
Executed instructions	1,521,770,857	1,379,585,371 *-9.34%*	1,201,446,771 *-21.05%*	1,323,863,372 *-13.01%*	1,243,176,562 *-18.31%*
CPI	2.290	2.449 *+6.94%*	2.463 *+7.55%*	2.866 *+25.15%*	2.844 *+24.19%*
Allocation latencies (includes `InitBlocks`)	1,992,397	159,771,419 *+7,919%*	2,019,380 *+1.35%*	872,365,975 *+43,684%*	947,655,295 *+47,464%*
Icache hits	1,781,285,257	1,593,341,822 *-10.55%*	1,383,456,346 *-22.33%*	1,486,054,805 *-16.57%*	1,382,595,042 *-22.38%*
Icache fetches	1,781,550,947	1,593,559,539 *-10.55%*	1,383,498,060 *-22.34%*	1,486,390,837 *-16.57%*	1,384,153,647 *-22.31%*
Icache hit rate	99.985%	99.986% *+0.001%*	99.997% *+0.01%*	99.977% *-0.008%*	99.887% *-0.10%*
Dcache hits	157,149,870	157,142,802 *-0.004%*	145,827,427 *-7.20%*	147,514,361 *-6.13%*	136,626,137 *-13.06%*
Dcache fetches	327,618,592	311,164,016 *-5.02%*	283,940,066 *-13.33%*	325,452,834 *-0.66%*	322,075,155 *-1.69%*
Dcache hit rate	47.967%	50.502% *+5.28%*	51.359% *+7.07%*	45.326% *-5.51%*	42.421% *-11.56%*
Arbiter operations	30,511,387	22,938,530 *-24.82%*	15,618,422 *-48.81%*	15,467,170 *-49.31%*	8,322,893 *-72.72%*
Latencies — Costs	134,734,847	173,383,249 *+28.68%*	60,754,829 *-54.91%*	109,597,538 *-18.66%*	61,536,566 *-54.33%*
Percent waste	3.867%	5.133% *+32.74%*	2.053% *-46.91%*	2.888% *-25.32%*	1.741% *-54.98%*
Cycles for GC	2,640,265	2,723,808 *+3.16%*	3,069,843 *+16.27%*	783,702,392 *+29,582%*	940,517,961 *+35,522%*
Fraction GC active	0.075%	0.081% *+8.00%*	0.104% *+38.67%*	20.655% *+27,440%*	26.603% *+35,371%*
Flips	3	3 *0%*	3 *0%*	1164 *+38,700%*	1381 *+45,933%*
Total allocations	47,741	47,741 *0%*	47,741 *0%*	7,622,586 *+15,867%*	7,622,803 *+15,867%*
Impeded allocations	0	0 *0%*	0 *0%*	125,283 *+∞%*	680,049 *+∞%*
Percent impeded	0%	0% *0%*	0% *0%*	1.644% *+∞%*	8.921% *+∞%*
Bus utilization	35.840%	39.217% *+9.42%*	41.095% *+14.66%*	48.950% *+36.58%*	48.883% *+36.39%*
Utilization for invalidation	1.836%	2.364% *+28.76%*	2.668% *+45.32%*	0.898% *-51.09%*	0.849% *-53.76%*

compiler represents only a slight improvement; but the SU version of `lisp` is over 22% faster than the base version. The performance of the heap-allocating compilers was mixed, being markedly worse on `sfft` than on `lisp`, but each heap-allocating compiler exhibited uniformly worse overall performance than the corresponding stack-based compiler, despite usually executing fewer instructions. This is more clearly shown by the CPI figures: the heap-allocating programs require an average of about 25% more cycles to execute each instruction than do the stack-based programs. What is the source of this delay?

The immediate cause is apparently cache performance. The instruction cache hit rates are slightly worse for the heap-allocating compilers than for the other three, and their data cache hit rates are far worse. There appear to be a few additional factors involved since, for the `lisp` cases, the HU compiler has a lower CPI than the HS compiler, despite the fact that HU's instruction and data cache hit rates are both lower than HS's. One possibility is that the total cache miss penalties for HU are less than those for HS. Future experiments will gather information on cache miss penalties to test this hypothesis. In any case, the CPI differences between the HS and HU compilers are insignificant.

The allocation latency figure for the SS and HS compilers should be viewed cautiously. The tables give the appearance that these compilers spend a great deal more time allocating objects than does the base compiler. The reason for this is that the SS compiler shares argument blocks between function calls, so that each function call requires an `InitBlock` operation to set the tag bits for the argument block. In the base compiler, this is accomplished with a `StackPush`. The allocation latency figure is calculated by adding up all the latencies for `Alloc*` and `InitBlock` operations, so this figure is artificially inflated for the compilers that share argument blocks. If the latencies for `InitBlock`s are subtracted from the values in Table 6.4, the allocation latencies for SS and HS become 1,668,103 and 678,867,440, respectively. These values are too low, however, since some of the `InitBlock`s are associated with allocations and should be counted. The truth lies somewhere between.

In any case, the important thing is that the "true" allocation latencies for the heap-allocating compilers are much greater than those of the stack-based compilers. This reflects the latencies incurred for allocating activation frames, but there is an

additional component to this difference. Because of the much greater need for garbage collection in the heap-allocating programs (see below), a number of allocations are *impeded*; that is, they must be delayed until the garbage collector has performed an amount of garbage collection proportional to the size of the allocation request. (Recall from section 5.4 that no allocations were found to be impeded in the experiments with the base compiler.) The HU compiler most strongly illustrates this problem; over 9% of all allocations in the `lisp` test cases were impeded, as opposed to only 1.7% for the HS compiler. Impedance has a large effect on allocation latencies, and hence on overall performance.

As mentioned, the amount of garbage collection activity for the heap-allocating compilers is much greater than for the stack-based compilers. For example, in the `lisp` test cases garbage collection was active one fourth of the time for the HS compiler, and one third of the time for the HU compiler. This is, of course, due to the rapid allocation and discarding of activation frames, which in large part become garbage almost as soon as they come into existence. The total number of allocated objects, and the number of flips of *from-space* and *to-space*, increase rapidly for these compilers as well.[4]

The compilers that don't share argument blocks waste far fewer cycles because of the difference between costs and latencies than do the stack-based compilers. As shown in the tables, this is because the former make fewer requests to the arbiter. A little thought shows why this should be the case. Consider the overhead due to communication with the arbiter to set up argument blocks for the function calls within a single function body. When argument blocks are not shared, this overhead consists entirely of a sequence of $\lceil n/32 \rceil$ `StackPush` operations, where n is the number of words in the argument blocks. When argument blocks are shared, on the other hand, each function call requires an `InitBlock` to initialize the tag bits to match the current arguments. If each function call within the function body is only executed once, the total number of words initialized by `InitBlock`s is n; but more than $\lceil n/32 \rceil$

[4]The HU compiler issues more allocation requests than the HS compiler because each allocation request that cannot be satisfied until a flip is performed must be reissued after the flip. The difference in the number of allocations equals the difference in the number of flips.

`InitBlock` operations are required whenever any of the argument blocks are not an exact multiple of 32 words, which is nearly always. Of course, many function calls will be executed more than once because they reside within loops, so the total number of `InitBlock`s can be expected to be much greater than the corresponding number of `StackPush` operations.

The heap-allocating compilers also make fewer requests to the arbiter than do the stack-based compilers. This is because the `lisp` test case in particular has a small average activation frame size; if this were not the case, the number of requests to the arbiter would be similar for both allocation methods. To see this, consider the arbiter operations required to allocate and release an activation frame. The stack-based compilers perform some number m of `StackPush` operations and one `StackPop`. The heap-allocating compilers perform an `AllocRec` operation and m `InitBlock` operations. However, if the activation frame size is at most 32 words, the heap-allocating compiler need only perform a single `AllocInitRec`, while the stack-based compiler must still perform both a `StackPush` and a `StackPop`. This accounts for the difference in the number of arbiter operations between the two allocation methods.

The heap-allocating compilers spend less than half as many cycles overall on cache invalidation, despite the heavy amount of garbage collection. This is because the dominant cause of cache invalidation is the `StackPush` operations so common in the stack-based compilers. Despite the drop in this component, however, the heap-allocating compilers have much higher overall bus utilization figures than their stack-based cousins. This is in large part due to the increased number of data cache misses they incur while waiting for the results of impeded operations. When more operations are impeded, the fraction of total executed instructions spent in the busy-waiting loop increases, resulting in a higher ratio of bus-busy cycles to total elapsed cycles.

In summary, the SU compiler has been shown to produce the best results for both of the programs studied here. Heap allocation of activation frames, as presently implemented, is inferior to stack allocation because of the enormous increase in garbage collection activity it causes; too-frequent collections result in impeded allocations, raising allocation latencies and bus activity. It is also best not to share argument

blocks among function calls, since separate argument blocks result in fewer arbiter requests than are required with separate argument blocks.

6.3 A solution to the stack manipulation overhead problem

None of the alternatives explored in the previous sections of this chapter provides acceptable performance. This is not surprising, since the activation frame bottleneck was not uncovered until after these experiments had been completed. Although the SU compiler was shown in the previous section to outperform the other alternatives, it suffers from the severe drawback of depending on the `StackPush/StackPop` protocol. The HS and HU compilers, which allocate their activation frames from garbage-collected memory, do not depend on this protocol; but their performance is worse than that of the SU compiler, because (1) they still incur a large amount of overhead at the beginning of each function call, and (2) they create garbage at a very high rate, requiring larger memory sizes and causing garbage collection to be active a high fraction of the time. This in turn increases the cost of allocations.

Another alternative that should be explored combines the best attributes of the SU and HU compilers and discards those attributes that are responsible for performance degradation. The SU compiler does not allocate a new object for each activation frame, and the HU compiler does not use the run-time stack object provided by the arbiter; these choices have been shown to be superior. How can these two seemingly exclusive characteristics be combined?

One solution is to allocate activation frames from the heap, *but not at every function call.* The linker, in cooperation with the compiler, can generate an array of free list headers, one for each unique activation frame PLD. At the end of each function, that function's activation frame is placed on the appropriate free list, preventing it from becoming garbage. At the beginning of a function call, prologue code first attempts to obtain an activation frame from the appropriate free list. Only if that fails does the mutator request a new activation frame from the arbiter. This technique depends upon the principle of function call locality, which states that if a program has called a certain function recently, it is quite likely to call it again in the

near future. If the "activation frame hit rate" is sufficiently high, the overhead for each function call should decrease dramatically.

If free lists are never discarded, the amount of memory taken up by discarded activation frames may grow to become an unacceptable fraction of garbage-collected memory. To avoid this, free lists are discarded at the time of a flip. This is done as follows. The array of free list headers is addressed at all times by a dedicated register. At each flip, a new array of free list headers is allocated from the heap, and the address of the new array is stored in the dedicated register. The old array of free list headers thus automatically becomes garbage, as does every activation frame on any of the free lists. Although it seems reasonable to believe that occasionally discarding the free lists is advisable, the tradeoffs between keeping and discarding the free lists should be investigated.

6.3.1 Performance model of activation frame caching

This section is devoted to a "thumbnail analysis" of the gains that can be expected from this technique. This discussion focuses on comparing the function call overhead of the SU compiler with the expected function call overhead of a heap-allocating compiler that caches activation frames, referred to hereafter as the HC compiler. *Overhead* is measured in terms of the number of additional instructions executed. The present analysis examines only the `lisp` workload; changes in the analysis for `sfft` and `troff` are discussed in the notes that follow. The direct cost of flips is ignored in this discussion. The major effect of flips is to lower the probability that a recycled activation frame is available; this effect is factored into the probability p_{hit} defined below. The analysis also presumes that argument blocks are not shared.

Define the following quantities:

$$
\begin{aligned}
\Omega_{\text{HC}} &= \text{total function call overhead for the HC version of } \texttt{lisp} \\
p_{\text{hit}} &= \text{probability of an activation frame cache hit} \\
\omega_{\text{hit}} &= \text{average overhead per function call, cache hit} \\
\omega_{\text{miss}} &= \text{average overhead per function call, cache miss} \\
F &= \text{total number of function calls executed}
\end{aligned}
$$

These parameters are related as follows:

$$\Omega_{\mathrm{HC}} = [p_{\mathrm{hit}}\,\omega_{\mathrm{hit}} + (1 - p_{\mathrm{hit}})\,\omega_{\mathrm{miss}}]F$$

To calculate the values of ω_{hit} and ω_{miss}, it is necessary to exhibit an efficient implementation of the proposed function call mechanism. Recall from section 6.1 that the heap-allocating compilers allocate an extra word of memory for each activation frame, in order to keep the frame pointer (r30) from referencing another object (see Figure 6.3). It is convenient to use this additional word to construct the free lists for discarded activation frames; that is, if a frame is on a free list, this word points to the next frame on the list, if any. Assuming that register r14 contains the address of the array of free list headers, that the free list header applicable to the current function is located at offset `N` from the beginning of this array, and that the activation frame is `F+4` bytes in length, the following code fragment will allocate an activation frame for the current function. The mutator first tries to allocate a frame from the free list; if this fails, it requests the frame from the arbiter, using the protocol employed by the HS and HU compilers.

```
        lw    r25,N(r14)
        bnez  r25,Found
        nop
        ...allocate a new frame...
        j     Continue
        nop
Found:
        lw    r31,F(r25)
        sw    N(r14),r31
Continue:
```

Whenever a function exits, it must return its activation frame to the appropriate free list. This is done with the following code sequence.

```
        lw    r31,N(r14)
        sw    0(r30),r31
        addi  r31,r30,#-F
        sw    N(r14),r31
```

So each function must execute at least five additional prologue instructions and four epilogue instructions. Thus $\omega_{\text{hit}} = 9$ and $\omega_{\text{miss}} = 9 + \omega_{\text{alloc}}$, where ω_{alloc} is the number of instructions executed when allocating a new frame according to the HS/HU compiler protocol. Now there are two ways of allocating a new frame, depending upon the size of the frame. If the frame size is no more than 32 words, the mutator requests a new frame by sending an `AllocInitRec` request to the arbiter, specifying the size of the record and the tag bits for it. Otherwise, the mutator allocates the frame with an `AllocRec` request, and initializes the tag bits by sending a sequence of `InitBlock` requests to the arbiter, each specifying up to 32 tag bits. Define

$$\begin{aligned} p_{\text{small}} &= \text{probability that an activation frame contains less than 32 words} \\ \omega_{\text{small}} &= \text{number of instructions required to allocate a small frame} \\ \omega_{\text{large}} &= \text{number of instructions required to allocate a large frame} \end{aligned}$$

Then

$$\omega_{\text{alloc}} = p_{\text{small}}\,\omega_{\text{small}} + (1 - p_{\text{small}})\,\omega_{\text{large}}.$$

When the activation frame is no larger than 32 words, the mutator places the size of the frame in register `r4` and the tag bits in register `r5` before calling the library routine `__gc_alloc_init_rec`, as shown in the following example:

```
addi  r4,r0,#86
lhi   r5,#27
jal   __gc_alloc_init_rec
addui r5,r5,#40963
```

The number of instructions executed in `__gc_alloc_init_rec` is $10+8n$, where n is the number of iterations of the busy-waiting loop required to complete the allocation. Note that the busy-waiting loop for allocations is longer than that for the stack manipulation operations discussed in chapter 5. This is because if the result of an allocation request is zero, the mutator must read the `GCStatus` port to see whether this is because the arbiter has not yet completed the operation, or because it is time for a flip. Combining the above code fragment with the `__gc_alloc_init_rec` subroutine gives $\omega_{\text{small}} = 14 + 8n$.

To calculate the value of n requires estimating the average latency of an `AllocInitRec` operation. Since the heap-allocating compilers analyzed above experienced impeded allocations because of their high rates of garbage collection, and since the new mechanism is anticipated to have garbage collection rates closer to those of the stack-based compilers, it is most appropriate to use the latency figures gathered for the SU compiler to estimate n. The following results are extracted from the raw data in reference [44].

Total `AllocInitRec` latencies	1,952,775 cycles
Number of `AllocInitRec`s	46,656
Average `AllocInitRec` latency	41.85 cycles
Average instructions per `AllocInitRec`	15.82

The average number of instructions listed above is computed by dividing the average latency by the average CPI for `lisp` of 2.645 (see Table 6.3). Now $n = \lceil 15.82/8 \rceil = 2$, so $\omega_{\text{small}} = 30$.

Consider next the case where the frame size exceeds 32 words. Then each function call executes code such as the following:

```
jal   __gc_alloc_rec
addi  r4,r0,#252
lhi   r27,(L5510>>16)&0xffff
jal   __gc_init_block_loop
addui r27,r27,(L5510&0xffff)
```

Here the mutator requests a record of 252 bytes from the arbiter, specifying the address of a PLD for the activation frame in register `r27`. In this case the PLD was located at the address labeled `L5510`.

The `__gc_alloc_rec` subroutine executes $9 + 8m$ instructions, where m is the number of iterations of the busy-waiting loop required before the result is returned. The `__gc_init_block_loop` subroutine, which issues the necessary sequence of `InitBlock` requests, requires $13+(12+4n)I$ instructions, where n is the average number of iterations of the busy-waiting loop required for each `InitBlock`, and I is the average number of `InitBlock` requests. Then

$$\begin{aligned} \omega_{\text{large}} &= 5 + (9 + 8m) + [13 + (12 + 4n)I] \\ &= 27 + 8m + (12 + 4n)I \end{aligned}$$

Now function activation frames tend to be relatively small, so it is justifiable to estimate this expression using a fairly small value of I.[5] Assuming that $I = 2.5$, corresponding to an average activation frame size of 320 bytes, gives

$$\begin{aligned} \omega_{\text{large}} &= 27 + 8m + (12 + 4n)(2.5) \\ &= 57 + 8m + 10n \end{aligned}$$

Once again it remains to estimate the latencies of the `AllocRec` and `InitBlock` operations, in order to compute m and n. Using the SU data from reference [44]:

Total `AllocRec` latencies	20,529 cycles
Number of `AllocRec`s	1077
Average `AllocRec` latency	19.06 cycles
Average instructions per `AllocRec`	7.206
Total `InitBlock` latencies	27,532 cycles
Number of `InitBlock`s	1165
Average `InitBlock` latency	23.63 cycles
Average instructions per `InitBlock`	8.934

This gives $m = \lceil 7.206/8 \rceil = 1$ and $n = \lceil 8.934/4 \rceil = 3$, from which $\omega_{\text{large}} = 95$. Thus

$$\begin{aligned} \Omega_{\text{HC}} &= [p_{\text{hit}}\,\omega_{\text{hit}} + (1 - p_{\text{hit}})\,\omega_{\text{miss}}]F \\ &= [9p_{\text{hit}} + (1 - p_{\text{hit}})(9 + \omega_{\text{alloc}})]F \\ &= [9p_{\text{hit}} + (1 - p_{\text{hit}})[9 + p_{\text{small}}\,\omega_{\text{small}} + (1 - p_{\text{small}})\,\omega_{\text{large}}]]F \\ &= [9p_{\text{hit}} + (1 - p_{\text{hit}})[9 + 30p_{\text{small}} + 95(1 - p_{\text{small}})]]F \\ &= [9p_{\text{hit}} + (1 - p_{\text{hit}})(104 - 65p_{\text{small}})]F \end{aligned}$$

For the combined `lisp` workload, $F = 7,426,251$. To estimate p_{small}, consider that the ratio of small activation frames to large activation frames should be roughly equal

[5] Criticisms may be leveled at the methods used to estimate ω_{large}. However, the suspicious reader may verify that increasing the values of m, n, and I has a minimal impact on the results of this analysis.

to the ratio of `AllocInitRec` operations to `AllocRec` operations taken from the HU data in reference [44].[6] This gives

$$\begin{aligned} \text{Number of } \texttt{AllocInitRecs} &= 7,108,255 \\ \text{Number of } \texttt{AllocRecs} &= 175,431 \\ p_{\text{small}} &= 0.9759 \end{aligned}$$

Thus

$$\begin{aligned} \Omega_{\text{HC}} &= [9p_{\text{hit}} + (1 - p_{\text{hit}})(104 - 65(0.9759))](7,426,251) \\ &= [9p_{\text{hit}} + (1 - p_{\text{hit}})(40.57)](7,426,251) \\ &= (40.57 - 31.57p_{\text{hit}})(7,426,251) \end{aligned}$$

Unfortunately, there is no empirical evidence on which to base an estimate of the activation frame hit rate. However, it seems reasonable to believe that activation frame hit rates will be very high in most programs; choosing $p_{\text{hit}} = 0.95$ is likely to be conservative. Table 6.5 calculates the total function call overhead for several values of p_{hit}. The rightmost column in this table shows the expected percent increase in executed instructions when moving from a traditional architecture to a garbage-collection architecture that uses the proposed function call mechanism.

Table 6.5: Expected overhead of proposed function call mechanism

p_{hit}	Ω_{HC}	% Overhead
0.99	69,180,726	20.66%
0.95	78,558,596	23.46%
0.90	90,280,933	26.96%
0.75	125,447,945	37.47%
0.50	184,059,631	54.97%

[6] The ratios are not exactly equal, because some allocated objects are not activation frames. However, the number of activation frames far outweighs the number of other allocated objects, so this ratio provides a good estimate.

How does this compare with the function call overhead of the SU compiler? A rough calculation of SU's overhead on the `lisp` workload can be made as follows. Using the data in reference [44], the percent of SU's arbiter operation latencies due to stack manipulations can be calculated:

Total latencies for `StackPush/Pop` (cycles)	591,601,686
Total latencies for arbiter operations (cycles)	593,603,989
Percent attributable to stack operations	99.66%

The number of instructions of function call overhead can then be calculated as follows:

Total instructions executed, SU compiler	852,095,631
Total instructions executed, no GC	334,839,226
Excess instructions	517,256,405
Percent attributable to function calls	99.66%
Function call overhead, SU	**515,497,733**
Percent overhead	153.95%

Clearly, the proposed mechanism provides performance far better than that provided by the SU compiler, which was shown to be the best of the alternatives explored in section 6.2. Table 6.6 illustrates this by showing, for several values of p_{hit}, the percent of the SU overhead that can be eliminated by employing the proposed function call mechanism.

Table 6.6: Improvement of proposed mechanism over SU compiler

p_{hit}	Ω_{HC}	Ω_{SU}	% overhead eliminated
0.99	69,180,726	515,497,733	86.58%
0.95	78,558,596	515,497,733	84.76%
0.90	90,280,933	515,497,733	82.49%

In summary, this analysis has shown that caching heap-allocated activation frames should produce much better performance figures than were obtained for the simpler function call mechanisms explored in section 6.2. Provided that activation frame hit rates are high, this method should produce code for `lisp` that executes no

more than 20–25% more instructions than a traditional program using `malloc()` and `free()`. Although the amount of improvement should not be considered to apply directly to the `troff` and `sfft` test cases, it is certain that their performances will also be improved by this technique. The general framework of the foregoing analysis applies, but in particular the values of p_{small}, ω_{small}, and ω_{large} may differ somewhat. The performance improvements for the two other programs will probably not be quite as dramatic as for `lisp`, but this is not unexpected, since their performance did not show as large a decrease in the chapter 5 experiments as did `lisp`.

It should be emphasized that the preceding analysis determines overhead only in terms of instructions executed. This is obviously not the only component of overall performance. In particular, it has been seen that CPI increases when the garbage-collection architecture is used. The following section contains the results of some preliminary experiments that appear to validate the predictions of the model, and provide insight into achievable overall performance.

6.3.2 Preliminary experimental results

The HC protocol described in section 6.3 has recently been implemented. Insufficient results have yet been gathered on which to base firm conclusions, but two representative simulations have been completed. This section reports on the results of these simulations.

In addition to employing the HC protocol, the compiler used in these experiments differs from previous versions in two ways. First, the partial invalidation protocol described in section 5.5.1 was implemented. This technique provides a smaller improvement than was measured in that section, since the HC compiler makes no use of `StackPush` calls; but it still reduces the overhead of the less-frequent `InitBlock` operation. Second, the busy-waiting loop has been removed for all operations that read from the `GCResult` register. These operations include all allocation requests and the `TendDesc` service. Instead, the mutator is stalled upon reading the `GCResult` register until the register contains the result of the pending operation. In informal experiments, this has been found to produce slightly better overall performance than the busy-waiting loop.

Tables 6.7 and 6.8 contain the results of running the `sfft` and `lisp` test cases,

Table 6.7: Results of HC experiment, `sfft/small`

Statistic	**No GC**	**GC/HC**	**Change**
Elapsed cycles	99,204,183	116,044,186	+16.98%
Executed instructions	61,071,302	61,389,042	+0.52%
CPI	1.624	1.890	+16.38%
Allocation latencies	6,602	39,683	+501%
Icache hits	63,679,752	64,064,290	+0.60%
Icache fetches	63,685,978	64,079,782	+0.62%
Icache hit rate	99.990%	99.976%	-0.01%
Dcache hits	21,490,727	21,743,196	+1.17%
Dcache fetches	21,558,670	21,818,937	+1.21%
Dcache hit rate	99.685%	99.653%	-0.03%
Latencies−Costs	0	6,607	$+\infty$%
Percent waste	0%	0.0057%	$+\infty$%
Cycles for GC	0	0	0%
Fraction GC active	0%	0%	0%
Bus utilization	27.706%	39.873%	+43.91%
Utilization for invalidation	0%	0.002%	$+\infty$%

respectively, on one set of input data each. It is apparent that the model of the previous section predicted very closely the behavior of the `lisp` test case. Recall that, for the combined `lisp` test cases, the increase in the number of instructions due only to the function call mechanism was predicted to be less than 24% when the activation frame hit rate p_{hit} is greater than 95%. The measurement for one of these test cases indicates that the increase in the *total* number of instructions, whether or not due to the function call mechanism, is approximately 21%. For the `sfft` test case, the number of instructions increased by only one half of one percent.

It turns out that p_{hit} can be calculated using the data collected from previous experiments. The total number of function calls can be found by looking up the number of `StackPop` operations executed by the corresponding test cases compiled using the SU compiler. By comparing the number of allocated records between the SU and HC trials, the number of allocated activation frames can be determined. It is then straightforward to find p_{hit}. Table 6.9 shows the calculation of p_{hit} for the two test cases analyzed here.

Table 6.8: Results of HC experiment, `lisp/prune`

Statistic	No GC	GC/HC	Change
Elapsed cycles	185,745,915	398,277,358	+114%
Executed instructions	118,421,396	143,619,161	+21.28%
CPI	1.569	2.773	+76.74%
Allocation latencies	5,061,208	1,090,383[a]	-78.46%[a]
Icache hits	131,965,860	159,537,718	+20.89%
Icache fetches	132,038,195	159,553,254	+20.84%
Icache hit rate	99.945%	99.990%	+0.045%
Dcache hits	27,158,540	36,928,597	+35.97%
Dcache fetches	27,333,214	37,143,213	+35.89%
Dcache hit rate	99.361%	99.422%	+0.06%
Latencies−Costs	0	107,322[a]	+∞%
Percent waste	0%	0.027%[a]	+∞%
Cycles for GC	0	1,697,336	+∞%
Fraction GC active	0%	0.43%	+∞%
Bus utilization	33.602%	66.806%	+98.82%
Utilization for invalidation	0%	0.004%	+∞%

[a] Some latencies were estimated.

Given a p_{hit} value of 0.9998, the model of the previous section predicts that the number of instructions executed for the combined `lisp` workload should increase by 19.97%. The slightly higher value of 21.28% shown in Table 6.8 includes instruction overhead for the other arbiter calls as well, so the prediction of the model appears to be quite accurate. This is in no sense a formal validation of the model, but it may perhaps serve to increase one's faith in its predictions.

In any case, the increase in the number of instructions executed has been reduced by the HC protocol to acceptable levels. However, overall performance is still quite a

Table 6.9: Activation frame hit rates

Trial	SU `StackPops`	SU allocations	HC Allocations	Activation frames	p_{hit}
`sfft/small`	68,889	2	37	35	0.9995
`lisp/prune`	2,554,723	18,798	19,345	547	0.9998

bit worse for the garbage-collected architecture than for the traditional architecture. Despite the modest increase in instruction counts, total elapsed time increases 114% for `lisp` and 17% for `sfft`. Clearly the average number of cycles required to execute an instruction is much higher for the garbage-collection architecture than for the traditional architecture. Apparently solving the function call overhead problem has uncovered a new bottleneck in the system. What is the new source of performance degradation?

It is apparent that cache hit rates are not the problem. Both the instruction and data cache hit rates drop only slightly for the `sfft` test case as the garbage-collection architecture is introduced, and they actually *increase* modestly for the `lisp` test case. Observe, however, the increase in bus utilization. The bus is actually busy almost twice as much of the time for the garbage-collection architecture as for the traditional architecture when running the `lisp` test case. Thus despite the increase in cache hit rates, the *cost* of an average cache miss is much higher. For the garbage-collection architecture, the data cache miss cost for `lisp` is over 34 processor cycles. Unfortunately, the costs of cache misses were not measured for the traditional architecture. When bus utilization is low, however, other test cases show that a data cache miss generally costs about 7 cycles. Clearly the high bus utilization is a source of severe degradation.

But what causes the bus to be so heavily utilized? Detailed tracing of the simulator shows that the bus is becoming saturated primarily because of write traffic. Recall that the original design of the hardware calls for data coherence to be maintained through the use of a write-through cache, with the garbage collector invalidating cache lines that it is going to modify. This means that every modification results in a write to the slow main memory, thus tying up the bus for an average of about five CPU cycles. While the bus is busy with write traffic, it cannot be used to fetch uncached instructions or data operands from main memory. This causes the DLX pipeline to stall until the bus is free, resulting in higher CPI.

Knowing that bus utilization is high because of write traffic does not explain why the garbage-collection architecture suffers more heavily from this phenomenon than does the traditional architecture. The explanation for this probably lies in the function prologue and epilogue code. The HC compiler produces a minimum of

five more store instructions per function call than does the compiler targeted to the traditional architecture. Three of these are required to manipulate the activation frame free lists; one is used to save the parent argument pointer register; and the last is needed to temporarily save the return address register before allocating the activation frame. Altogether the HC version of `lisp` executes about 12.8 million more store instructions than does the traditional version. Thus the write-through cache provides much greater penalties to the garbage-collection architecture with the HC protocol than it does to the traditional architecture. Section 6.3.3 outlines a new approach to the data coherence problem that does not require the use of a write-through cache.

Even with this modification, however, the garbage-collection architecture can be expected to exhibit somewhat higher bus utilization than the traditional architecture. This is because of the run-time library routines that perform uncachable stores to, and fetches from, the arbiter ports. Since the total number of such communications is relatively small when the HC protocol is used, this should not have a large effect on bus utilization, and thus on overall CPI.

Because of a deficiency in the stall-on-result-fetch implementation of the simulator, the simulator failed to report the latencies of certain operations. These latencies were estimated by adding 5 cycles to the corresponding costs.

6.3.3 A new approach to data coherence

Recall that it is necessary to maintain coherence between the data in garbage-collected memory and copies of that data that reside in the processor's data cache. Otherwise the garbage collector might copy an object from *from-space* into *to-space* without being aware that a copy of the object in the data cache has been modified. The current implementation of the simulator handles data coherence in a somewhat "brute-force" fashion. That is, whenever the CPU performs a store, the new data value is written to the slower main memory as well as to the faster cache. If the target address does not reside in the cache, furthermore, the cache is not written to. The garbage-collected memory module is responsible for initiating cache invalidation requests whenever it performs an operation that may modify a location contained in the cache. Using the partial cache invalidation strategy of section 5.5.1 and the

HC protocol of section 6.3, this only becomes necessary during `CopyBlock` operations and when a flip occurs.

It is clear why the `CopyBlock` service requires invalidation of the cache; the target locations of the `CopyBlock`, if cached, will no longer contain valid data. It is not so obvious why the arbiter invalidates all of *from-space* at a flip. After all, since the mutator is only permitted to have pointers into *to-space*, the processor should not be reading from cached *from-space* addresses anyway. However, it is possible for such an address to survive in the cache through an additional flip. At that point, the cached address again points to *to-space*, but contains data that is not coherent with the "true" copy in garbage-collected memory. To avoid this scenario, the arbiter must invalidate *from-space* addresses at a flip.

It is easy to see that much of the write traffic of the existing method is unnecessary. It is likely that a high percentage of the writes to garbage-collected memory are not needed, since before the next flip it is probable that the data will either become garbage or be overwritten. With a *write-back* cache policy, in which a cache line is flushed to memory only when it is replaced, it should be possible to avoid most of this write traffic.

If a write-back cache is used, however, it is still necessary to ensure that data coherence is maintained. For this purpose, the new method assumes that the CPU's data cache is equipped with the ability to snoop the bus (that is, to monitor requests on the bus) and to respond to a "read-with-intent-to-modify" (RWIM) signal. This signal, which exists in many standard multiprocessor cache-coherence protocols (see, for example, [34, 52]), causes the cache to flush its copy of a cache line to the bus and mark its copy *Invalid*. The memory arbiter can use this protocol to ensure data coherence, as follows.

The garbage collector only needs to be assured of data coherence when it is copying an object from *from-space* into *to-space*, or when servicing a `CopyBlock` request. In either case, any word of the source object that resides in the cache must be flushed to garbage-collected memory before it is copied, and any word of the target object that resides in the cache must be invalidated. The garbage collector ensures the former action by issuing a RWIM signal for each source address before copying it. If the cache contains a copy of the data at the location specified by the RWIM, it

flushes it to the bus and marks it invalid. If the cache does not respond, the garbage collector detects this rapidly, since caches can respond to read requests much more quickly than main memory. It may then assume that the copy in garbage-collected memory is up-to-date. After broadcasting an invalidate signal for the target address, it may perform the copy.

Better performance can be achieved for `CopyBlock`s if the data cache supports mutator control over cache invalidations. (An example of a cache with this capability is described in reference [34].) In this case the mutator can issue the `CopyBlock` request and then invalidate the target addresses before reading the `GCStatus` register to see if the `CopyBlock` has completed. This increases efficiency in three ways: (1) the bus is freed from the burden of invalidation requests; (2) the garbage collector requires less time to service the `CopyBlock`; and (3) the mutator spends less time saturating the bus in the busy-waiting loop while waiting for the service to complete.

The new data coherence protocol has not yet been implemented, but it is expected to reduce bus utilization for the garbage-collection architecture to values nearer those of the traditional architecture. Program execution times on the two architectures should then be much more similar than is the case in the results presented here. The `lisp` program, which exhibits the worst execution time differences, is expected to take less than 30% longer to execute on the garbage-collection architecture than on the traditional architecture. The other programs in the workload will exhibit still lower penalties. Such performance figures would be quite acceptable to those applications requiring fine-grained real-time garbage collection.

7. LANGUAGE EXTENSIONS TO SUPPORT SLICE OBJECTS

Many real-world problems employ arrays whose elements have different lifetimes. The most obvious examples of such arrays are *strings* and *streams*. Strings are finite sequences of character data, while streams are unbounded sources or sinks of any type of data. Pattern-matching programs often read large quantities of string data and discard those portions that do not match particular patterns. Similarly, streams can be used to implement I/O with files, devices, and cooperating processes, or to generate infinite sequences of data algorithmically; thus streams can process large quantities of data with limited and varying lifetimes. Unlike strings, which are usually considered to be arrays of characters, streams may be composed of any element type. For example, a program might use two input streams containing frames of audio and video data, and combine them to create an output stream of synchronized audiovisual frames. Or a stream might embody sample data from chemical processes gathered periodically in real time.

The slice objects discussed briefly in section 2.2 provide a convenient mechanism for implementing arrays of this nature. Slices as a part of a real-time garbage collection system have been studied [36, 38] in the context of the programming language Icon [14]. However, this work used only software methods, and consequently programs using this garbage collection scheme ran two to three times slower than the same programs using the original Icon run-time library. Thus it is of interest to know how slices perform using the hardware-assisted garbage collection algorithm.

Unfortunately the C++ programming language has no direct support for slice objects. The compilers discussed in the foregoing chapters allocate all arrays as record objects, meaning that the lifetimes of all elements of an array are identical. Thus even if only one element of an array is still needed at some point in program execution, the space for the entire array must be retained. Subslices as such do not

exist in C++. Clearly memory utilization could be improved for many programs if slice objects were available in the language.

This chapter details some minor extensions to C++ that permit programmers to utilize the advantages of slice objects. As a demonstration of their use, a simple line editor has been developed based on a `String` class implemented using slices. The performance of this editor is compared with that of the same editor using `String` classes from two widely available class libraries. The results of this study are discussed in section 7.3.

7.1 Syntax and informal semantics

Before discussing the extensions themselves, it is appropriate to outline the features that programmers would need when manipulating slices.

- It should be possible to declare a slice object of any element type. Recall, however, that slice objects are initialized by the garbage collector and are unwritable by the mutator. Thus the declaration of a slice object should not permit direct manipulation of the slice object itself, but should provide a syntax for reading and modifying the slice data region referenced by the slice object.

- A construct must be provided to allow the programmer to allocate a slice object referencing any number of elements.

- Programmers must be able to retrieve or modify a single element of slice data, and be able to create a subslice of an existing slice.

- The length of a slice object should be available to the programmer.

- Assignment to variables having a slice type should work in the usual way.

- It should be straightforward to concatenate the slice data from two slices into a single slice.

These characteristics have been implemented in the following constructs.

7.1.1 Declarations

Let *type* be a fundamental type or a user-defined type (a `class`, `struct`, or `union`). Then the declaration

`slice` *type identifier-list*;

is legal and declares each identifier in *identifier-list* to be a variable of type "pointer to slice of element type *type*." The type "slice of element type *type*" is a hidden type in the sense that no objects of this type can be declared explicitly, and the fields of this type are not directly accessible by the programmer.[1] Each identifier declared as above causes storage for a pointer to be reserved in the current scope, and binds the identifier to this location in the usual way.

Since declared slice objects are really pointers, the usual pointer operations (assignment, arithmetic, and comparison) are automatically applicable to them. The standard declarator operators can also be applied to slice declarations; thus the declarations in Table 7.1 are all legal. Slice declarations may also include storage-class specifiers (such as `extern`, `static`, and `register`) and type qualifiers (such as `const` and `volatile`).

7.1.2 Expressions

Let `id` be a variable of type "pointer to slice of element type *elttype*." Then the syntactic units in Table 7.2 represent legal expressions recognized by the extended C++ compiler. The appearance of `id` in an expression denotes the value of `id` in the usual way, i.e., the address of the current slice object addressed by `id`. The element selection operator `[]` for arrays is overloaded to have a similar meaning for slices: `id[`*expr*`]` denotes the *expr*th element (using zero-based indexing) of the slice region data referenced by the slice object addressed by `id`. An expanded element selection notation is used to represent the *subslice* operation: `id[`*expr1*`:`*expr2*`]` denotes a pointer to a subslice of `id` including the *expr1*th through *expr2*th elements

[1] Of course, a determined programmer can always cast a slice pointer into a pointer to a different type and manipulate the fields directly. However, any attempt to write to the slice object will be ignored by the hardware.

Table 7.1: Examples of slice declarations

Declaration	**Meaning**
`slice char x[4];`	Declare `x` as an array of four pointers to slice objects of element type `char`
`slice int *y;`	Declare `y` as a pointer to a pointer to a slice object of element type `int`
`slice Frame& z;`	Declare `z` as a reference to a pointer to a slice object of element type `class Frame`
`slice double f();`	Declare `f` as a function returning a pointer to a slice object of element type `double`
`slice char Box::* p;`	Declare `p` as a pointer to a member of `class Box` having type "pointer to slice object of element type `char`"

of `id`'s slice region data. That is, the `[:]` operator causes a new slice object to be created that references a subarray of the slice region data belonging to the parent slice. Allocation of a new, uninitialized slice object is specified using a variant of the `new` operator: `new slice` *elttype* `[`*expr*`]` causes a new slice object to be allocated that references slice region data containing *expr* elements of type *elttype*.

Table 7.2: Expression syntax for slice operations

Expression	**Informal semantics**
`id`	Value of `id`
`id[`*expr*`]`	Element selection operation
`id[`*expr1*`:`*expr2*`]`	Subslice operation
`id[]`	Length operation
`new slice` *elttype* `[`*expr*`]`	Slice allocation

The preceding paragraph actually contains a small untruth. The slice region data for each slice object actually contains space for one additional object of the element type, for reasons discussed (in regard to arrays) in section 4.1.3. The compiler

automatically generates code to allocate the extra element for newly allocated slices, or to include the extra element when creating a subslice of a preexisting slice.

The extensions outlined here were designed so that access to slice region data is as convenient, and nearly as efficient, as access to elements of standard arrays. For example, the code in Figure 7.1, which steps through the elements of a zero-terminated slice of integers, resembles almost exactly the code to step through the elements of a zero-terminated array of integers. Only the initialization portion of the `for` statement is slightly less efficient in the case of slices, where the expression `&x[0]` requires an additional dereference of a pointer. (Note that it would not be legal to write `ip = x` as one can with arrays, since `ip` and `x` here have different types.)

```
slice int x;
int *ip;

for (ip = &x[0]; *ip; ip++)
  printf("%d\n", *ip);
```

Figure 7.1: Example slice code

7.1.3 Possible extensions

It would be advantageous to allow constructs like the following (here `x` and `y` are pointers to slice objects of the same element type):

```
x[2:5] = y[1:4];
```

The meaning of this construct would be to copy elements 1 through 4 of `y`'s slice region data into positions 2 through 5 of `x`'s slice region data. Since all slice region data resides in garbage-collected memory, such expressions could be compiled into `CopyBlock` arbiter calls. In programs that do a lot of copying between slices, this would significantly reduce the load on the system bus.

It would be relatively straightforward to overload the assignment operator to have this revised meaning in the context of slice arguments, despite the fact that

`x[2:5]` (in our example) is not a legitimate lvalue. However, this in itself would not produce the best code. The rvalue `y[1:4]`, as mentioned above, is compiled into a subslice operation, resulting in the creation of a new slice object and returning a pointer to it. But this new slice object would immediately become garbage, since its address is not stored anywhere. It would be preferable, within the context of this construct, to skip the generation of the subslice operation, instead retaining the indices delimiting the source for the copy operation. Implementation of this was considered to be beyond the scope of this investigation.

The expressions described in section 7.1.2 have sufficient power to express the concatenation of two slice objects, but this is somewhat tedious for the programmer. He or she must find the lengths of the two source slices, allocate a slice whose length is the sum of these lengths, and copy the data into the new slice by brute force. Of course, it is easy to abstract this process away by wrapping a slice type in a class definition and creating a concatenation function for it. It would be reasonably simple to add a concatenation operator to the language to prevent having to do this for each class. Provided the two arguments to the operator are slice objects of the same element type, the concatenation operator would generate code to allocate a new slice object as discussed above. Instead of element-by-element copying, however, the compiler would generate calls to the `CopyBlock` arbiter primitive.

7.2 Implementation notes

As one would expect, it was quite simple to modify the GNU C++ lexer and parser to add the new constructs to the language. The new syntactic forms for expressions were added to the rules for recognizing expressions. A single additional lexeme (`slice`) was added to the lexical analyzer as a reserved word. The `slice` lexeme was added to the list of declaration modifiers, or "declmods"; declaration modifiers are keywords (such as `register`, `extern`, `const`, and `volatile`) used to alter or elaborate the meaning of a declaration. When the compiler has determined the basic type `T` of a declaration, it checks to see if its declmods include `slice`. If so, the type is converted to "pointer to slice of element type `T`."

Upon recognizing such a slice declaration, the compiler checks to see if a slice

object type with element type `T` has previously been built. If so, the existing type is used; otherwise a new one must be created. A slice object type is a record that contains two fields and has type name `$$slice`. The `$$length` field is an integer holding the read-only length of the slice, while the `$$data` field has type "pointer to `T`" and contains the read-only address of the slice region data. When the slice object type is created, a new pointer type that addresses the slice object type is also built. Thus for any element type `T`, there will always be at most one type "slice of `T`" and one type "pointer to slice of `T`." Reusing preexisting slice type nodes ensures that type-checking of expressions involving slices will be performed correctly.

The run-time library was expanded by the addition of four subroutines that communicate with the slice-related arbiter ports. These are `__gc_alloc_dslice` and `__gc_alloc_tslice`, to allocate a descriptor slice or terminal slice, respectively; and `__gc_alloc_dsubslice` and `__gc_alloc_tsubslice`, to create a descriptor or terminal subslice. The slice allocation routines expect register r4 to contain the size in bytes of the desired slice region data, and return the address of the allocated slice object in register r25. The subslice allocation routines use r4 and r25 for the same purposes, but in addition expect r27 to contain the base address of the slice region data to be referenced by the subslice.

The bulk of the additional compiler work for slices occurs when generating code for the new expression constructs detailed in the previous section. The general approach used is to transform each syntax tree for a slice syntactic construct into a more detailed form (utilizing the fields of the hidden slice object type) that can be directly processed by preexisting code generation routines. Table 7.3 shows the original syntax of the slice constructs, followed by their transformed internal representations. (Assume that `id` has been declared as a pointer to a slice object of element type *elttype*, and that `index`, `hi`, `lo`, and `nelts` are all integer-valued variables.)

An occurrence of an isolated slice pointer variable requires no special handling. The syntax tree for a slice element selection operation is modified to include the hidden indirection via the `$$data` field. Similarly, the syntax tree for a slice length operation is expanded to select the hidden `$$length` field of the slice object. However, the raw length field contains the length of the slice data region in *bytes*, and includes the space for the extra hidden element discussed above; but the semantics of the

Table 7.3: Internal representation for slice operations

Expression	**Internal representation**
`id`	`id`
`id[index]`	`id->$$data[index]`
`id[]`	`id->$$length / sizeof(`*elttype*`) - 1`

`new slice` *elttype* `[nelts]`	`r4 = sizeof(`*elttype*`) * (nelts + 1),`
	`_gc_alloc_tslice(),`
	`r25`

	`_gc_alloc_dslice(&TYPE_PLD(`*elttype*`),`
	`  sizeof(`*elttype*`) * (nelts + 1),`
	`  nelts + 1)`

`id[lo:hi]`	`r4 = sizeof(`*elttype*`) * (hi - lo + 2),`
	`r27 = &(id->$$data[lo]),`
	`_gc_alloc_`*x*`subslice(),`
	`r25`

length operator is to produce the number of *elements* in the slice without counting the hidden element. Thus the expanded syntax tree for this operation divides `$$length` by the element size and subtracts one for the hidden object.

The remaining two operations require more complex elaborations. When allocating a new slice, the compiler first determines from *elttype* whether to allocate a descriptor slice object or a terminal slice object. It then generates code to call either `__gc_alloc_dslice` or `__gc_alloc_tslice`. The latter function uses the streamlined call sequence described in section 4.1.5, in which arguments and returned values are passed in registers and only caller-save registers may be used in the function body. On the other hand, `__gc_alloc_dslice` is more complex, since it must initialize the tag bits for variable-sized arrays of any element type, and thus requires the use of many more registers than are available in the caller-save set. For this function the standard call mechanism is used.

When allocating a terminal slice, the compiler first places the appropriate size value in register r4 and then generates a call to `__gc_alloc_tslice`. The value of the entire operation is the resulting contents of r25, where the called function returns

the address of the allocated slice. In order to replace the original syntax tree with these several operations, the compiler generates code such as would be generated for a C++ comma expression, as depicted in Table 7.3. Note that the size passed to the allocation subroutine is in bytes, and is generated by multiplying the size of one element by one greater than the number of elements required. Constant folding is used so that, in most cases, no arithmetic is required at run time to determine the number of bytes to allocate.

For descriptor slice allocation, the compiler generates a call that passes three arguments to `__gc_alloc_dslice`. The first of these is the address of the PLD stored with the element type's syntax tree (designated in Table 7.3 by `TYPE_PLD(`*elttype*`)`). The second argument is the total size of the slice region data, generated in the same manner as for terminal slices, and the last parameter is the number of elements requested (incremented by one). In contrast to terminal slices, the returned address of a descriptor slice is passed in the same manner as a normal function call return value (that is, it is returned in register r27, the pointer return value register).

Syntax tree elaboration for a subslice operation is similar to that which generates a call to `__gc_alloc_tslice`. In this case the compiler must also place the base address of the original slice data region in r27. This address is given by taking the address of the result of a slice element selection operation; i.e., in our example, we take `&(id->$$data[lo])`. The size parameter is again given by one plus the desired number of elements, multiplied by the size of a single element; constant folding is used here as well. The called subroutine in this case is either `__gc_alloc_dsubslice` or `__gc_alloc_tsubslice`.

7.3 Results of experiments

In order to test the performance of slice objects, a simple text editor was written in C++ by Craig VanZante. The editor operates in line mode, maintaining a visible cursor (represented by a carat) to mark the position within the current line. It provides the capability to read in files, insert and delete characters, insert and delete lines, search for a string, move the cursor through the file, and save the edited results. The editor relies upon the existence of a generic string class. Three versions of this

Table 7.4: Results of `editor` experiments (1 of 2)

Statistic	`RJS`	`libg++`	`slice`/**1 MB**
Elapsed cycles	333,642,089	322,849,863	322,475,404
Executed instructions	191,481,737	196,643,073	112,949,857
CPI	1.742	1.642	2.855
Allocation latencies	41,297,171	10,586,890	1,250,516[a]
Icache hits	204,614,096	215,486,622	117,962,429
Icache fetches	207,169,813	217,168,250	121,640,817
Icache hit rate	98.766%	99.226%	96.976%
Dcache hits	36,416,285	37,641,336	27,887,234
Dcache fetches	36,532,836	37,763,394	28,218,945
Dcache hit rate	99.681%	99.677%	98.825%
Latencies−Costs	0	0	158,560[a]
Percent waste	0%	0%	0.049%[a]
Cycles for GC	0	0	2,000,729
Fraction GC active	0%	0%	0.62%
Bus utilization	43.487%	41.400%	67.043%
Utilization for invalidation	0%	0%	0.005%

[a] Some latencies were estimated.

generic class were written; one of these was defined in terms of character slice objects, while the others utilize two popular public-domain string classes: the `RJS` library, written by Roland J. Schemers III while at Oakland University, and the `libg++` library written by Doug Lea. Both libraries are quite general and provide many functions for manipulating strings and substrings.

The editor was compiled and linked with each of the three string classes. The `RJS` and `libg++` versions were built using the compiler targeted to the traditional architecture, while the slice version was compiled with the HC compiler, modified to include the language extensions described above. Each of these versions was executed on the simulator using a small input set that repeatedly reads in a file and makes substantial changes to it. For comparative purposes, the `slice` version was run in three different sizes of garbage-collected memory. The empirical data collected from these experiments appears in reference [44], and is summarized in Tables 7.4 and 7.5.

Table 7.4 compares the two editors constructed using the public-domain libraries

with the version compiled using slices. The `slice` version was run using one megabyte of garbage-collected memory, which is the smallest amount in which it will run using the HC protocol. Notice that the total performance of the three programs is virtually indistinguishable, with `libg++` and `slice` running about 3% faster than `RJS`. It is interesting to note, however, that the components of overall performance are quite different for `slice` than for the public-domain versions. The `slice` version executes far fewer instructions: 70% fewer than `RJS`, and 74% fewer than `libg++`. However, `slice` has a much higher CPI figure than the other two, because of the bus utilization problem discussed in section 6.3.2. Use of a write-back cache should cause `slice` to run substantially faster than the others, even in a small garbage-collected memory.

Although some of the latency figures had to be estimated because of the simulator deficiency mentioned in section 6.3.3, it is clear that `slice` allocates objects much faster than `RJS` and `libg++`. It is interesting to note that `libg++` seems to require much less time to allocate objects than `RJS`; one may presume that this is because `libg++` performs fewer allocations, since both use the same `malloc()` and `free()` routines. For this test case, `libg++` apparently did a better job of reserving extra space for strings, thus avoiding reallocation upon later expansion.

There are no surprises in cache performance. All programs had very high instruction and data cache hit rates; `slice` had slightly lower hit rates than the others. The data cache hit rate is expected to be lower because of the uncachable reads from the `GCResult` and `GCStatus` registers, and because of the effects of the copying garbage collection algorithm on caching. Instruction cache hit rates are presumably lower because of slightly larger code size, leading to more cache line replacements. The phenomenon of high bus utilization for the garbage-collection algorithm is repeated in these experiments; the combination of slightly poorer hit rates and much higher miss penalties results in a CPI increase of 64–74% over the cases employing the traditional architecture.

Table 7.5 shows the effect of garbage-collected memory size on the `slice` version of the editor. As memory size increases, `slice` begins to significantly outperform `RJS` and `libg++`. Running in 4 MB of garbage-collected memory, `slice` is 30% faster than `RJS`, and 26% faster than `libg++`. Thus despite the high CPI induced by bus saturation, slice objects have been shown to increase performance of programs making

Table 7.5: Results of `editor` experiments (2 of 2)

Statistic	`slice`/**1 MB**	`slice`/**2 MB**	`slice`/**4 MB**
Elapsed cycles	322,475,404	282,252,476	256,726,986
Executed instructions	112,949,857	112,947,888	112,945,947
CPI	2.855	2.499	2.273
Allocation latencies	1,250,516[a]	1,023,803[a]	1,053,622
Icache hits	117,962,429	117,960,124	117,957,945
Icache fetches	121,640,817	121,638,404	121,636,026
Icache hit rate	96.976%	96.976%	96.976%
Dcache hits	27,887,234	27,888,184	27,889,210
Dcache fetches	28,218,945	28,218,654	28,218,370
Dcache hit rate	98.825%	98.829%	98.834%
Latencies−Costs	158,560[a]	159,079[a]	159,076
Percent waste	0.049%[a]	0.056%[a]	0.062%
Cycles for GC	2,000,729	1,501,191	0
Fraction GC active	0.62%	0.53%	0%
Bus utilization	67.043%	61.351%	56.759%
Utilization for invalidation	0.005%	0.003%	0%

[a] Some latencies were estimated.

heavy use of string manipulation. This performance gap is expected to increase with the new data coherence method outlined in section 6.3.3.

Note that bus utilization and CPI both vary inversely with the size of garbage-collected memory. This is likely due to the effect of cache invalidation during garbage collection, which uses bus cycles both for the invalidation requests themselves and for the resulting refetches of invalidated data. Cache hit rates and wastage due to the difference between costs and latencies all remain roughly constant.

In summary, this chapter has shown briefly that (1) it is quite simple to extend C++ to support slice objects, and (2) slice objects may be used to implement a string class that outperforms at least some traditional methods. The difference in performance is most marked when a large amount of garbage-collected memory is available, and is expected to become even more significant when a more efficient data coherence mechanism is employed.

8. CONCLUSIONS AND FUTURE WORK

This dissertation has demonstrated the practicality of constructing a real-time garbage collection architecture that is capable of performance approaching that of more conventional architectures. Part of this work involved an early design of the object space manager, which is a critical component of the garbage-collecting memory module. The OSM design contained here has been discarded in favor of a more efficient design that makes use of standard DRAM technology. The primary lessons of the early design are that the OSM must (1) provide only essential functions, and (2) achieve highly regular circuit design, in order to be manufactured at a sufficiently high density and low cost. The later design incorporates these lessons.

Although the initial results of experiments using a prototype compiler and simulator were disappointing, these experiments were instrumental in uncovering and correcting the sources of the performance problems. Experiments with different function call mechanisms demonstrated that both traditional stack allocation and heap allocation of activation frames result in unacceptable levels of overhead. Analysis of the reasons for this have led to a new method for allocating activation frames in which discarded frames are cached for later reuse. Preliminary experiments show that the new method is capable of providing performance less than twenty-five percent slower than that of a traditional architecture, provided that a more efficient data coherence mechanism is used. This is well within the limits of acceptability for an architecture that also provides guaranteed upper bounds on allocation latencies.

A final contribution of this dissertation is a small set of extensions to the C++ programming language to support slice objects. Slice objects, supported by the garbage collection architecture, embody the abstract notion of a fragmentable array type; an example of such a type is a string type supporting substring operations. Experimental results show that programs using slice representations for strings can

outperform traditional C++ string implementations, despite remaining inefficiencies in the data coherence scheme.

The most pressing future work in this area is to complete the reimplementation of the compiler and simulator to correct the problems exposed through the experiments described in this dissertation. The activation frame caching mechanism and the protocol to stall the mutator on allocation requests have both already been implemented. The final remaining step is to reinstrument the data coherence mechanism as described in section 6.3.3.

The next major effort will be to construct a hardware prototype of the garbage-collecting memory module. The experimental results in this dissertation are very promising, but it remains to demonstrate that the assumptions of hardware performance are realistic and attainable. Industry and government partners are currently being sought to help fund such an effort.

It would be useful to gather additional statistics in future experiments as well. The allocation rates of programs will be of great interest in determining whether the requirement of pacing allocation to match garbage collection rates can be relaxed. If the probability that the mutator can get ahead of the garbage collector is essentially zero, removing this restriction should result in lower allocation latencies and better overall performance.

A much more ambitious project for the future would be to examine the behavior of the garbage-collection architecture under multitasking and multiprocessing loads. The current protocols are specific to a uniprocessor environment and would have to be slightly altered for a multiprocessing architecture in order to avoid interference between two mutators in accessing the arbiter. Other than this, there is nothing in the design of the garbage-collecting memory module to prevent its use in a bus-based, shared-memory multiprocessing environment. It would be most interesting to see how many processors can be effectively supported by the garbage collector.

ACKNOWLEDGMENTS

Many people have contributed to the research reported in this dissertation, either directly or through their personal or financial support. I would particularly like to thank the National Science Foundation, who supported me for three years with a graduate fellowship that allowed me to focus my energies on research. NSF also supported this project under grant MIP-9010412 and by funding an undergraduate research assistant during the summer of 1992. The Iowa State Computer Science Department and Graduate College also provided me with additional support during my years in this program.

I am also grateful to the Free Software Foundation and its GNU project, without which I would have had no basis on which to build the compiler modifications described here, nor would I have had the `troff` test case available in the public domain. Jim Lathrop, Tim Budd, and Craig VanZante also contributed test cases. David Martin was very helpful in reestablishing security after a hacker broke into one of our primary research computers. Gary Leavens was kind enough to offer use of his hardware resources when crunch time came; we also made heavy use of the computing resources made available through Project Vincent, a joint effort of Iowa State University and Digital Equipment Corporation.

Very special thanks go to my advisor, Dr. Kelvin Nilsen, for his unflagging support and countless late nights of debugging. I sincerely appreciate the freedom he always gave me to follow my own inclinations, and the gentle prodding that kept me moving in a productive direction. I also want to thank my excellent friends Jim Lathrop and David Martin; they have played a large part in keeping me sane during the last year.

Finally, and most importantly, my inexpressible gratitude goes to my wife, Lori, and my two sweet daughters, Rebecca and Rochelle. Over the last four years they

have sacrificed quietly, living frugally while waiting for their husband and father to find some scant amount of time to talk and play with them. Their long wait is finally at an end.

BIBLIOGRAPHY

[1] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters* **25** (1987): 275–279.

[2] H. G. Baker, Jr. List processing in real time on a serial computer. *Communications of the Association for Computer Machinery* **21** (1978): 280–293.

[3] J. F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical Report TN-12, Digital Western Research Laboratory, 1989.

[4] P. B. Cohen. An introduction to CMOS design styles. *VLSI Design* (1984): 88–96.

[5] A. L. DeCegama. *Parallel Processing Architectures and VLSI Hardware*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[6] D. L. Detlefs. Concurrent garbage collection in C++. Technical Report CMU–CS–90–119, School of Computer Science, Carnegie Mellon University, May 1990.

[7] E. W. Dijkstra. After many a sobering experience, 1975. E. W. Dijkstra note EWD500.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation, 1975. E. W. Dijkstra note EWD496.

[9] D. R. Edelson. Dynamic storage reclamation in C++. Technical Report UCSC–CRL–90–19, Department of Computer and Information Sciences, University of California at Santa Cruz, 1990.

[10] J. R. Ellis, K. Li, and A. W. Appel. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988, 11–20.

[11] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[12] E. D. Fabricius. *Introduction to VLSI Design.* McGraw-Hill, New York, 1990.

[13] Andrew Ginter. Design alternatives for a cooperative garbage collector for the C++ programming language. Technical Report 91/417/01, Department of Computer Science, University of Calgary, January 1991.

[14] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, NJ. Second edition, 1990.

[15] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language.* Prentice Hall, Englewood Cliffs, NJ. Second edition, 1971.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[17] L. B. Hostetler and B. Mirtich. *DLXsim—A Simulator for DLX*, 1990. User manual. Ordering information may be found in reference [16].

[18] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Object-oriented real-time language design: constructs for timing constraints. *Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, October 1990, 289–298, Ottawa, Canada. ACM Press, New York.

[19] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc., New York, 1991.

[20] S. N. Kamin. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley, Reading, MA, 1990.

[21] G. Kane. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[22] D. E. Knuth. *The Art of Computer Programming, volume 1.* Addison-Wesley, Reading, MA, 1968.

[23] R. H. Krambeck, C. M. Lee, and H. S. Law. High-speed compact circuits with CMOS. *IEEE Journal of Solid-State Circuits* **SC-17** (1982): 614–619.

[24] L. Lamport. On-the-fly garbage collection: Once more with rigor. Technical Report CA-7508-1611, Massachussetts Computer Associates, Wakefield, Massachussetts, 1975.

[25] L. Lamport. Garbage collection with multiple processes: An exercise in parallelism. Technical Report CA-7602-2511, Massachussetts Computer Associates, Wakefield, Massachussetts, 1976.

[26] R. G. Larson. Minimizing garbage collection as a function of region size. *SIAM Journal on Computing* **6** (1977): 663–667.

[27] C. E. Leiserson. *Area-Efficient VLSI Computation.* MIT Press, Cambridge, MA, 1982.

[28] C. E. Leiserson. Fat-trees—universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers* **34** (1985): 892–901.

[29] E. Lewis. The design and performance of 1.25μ CMOS. *VLSI System Design* (1987).

[30] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* **26** (1983): 419–429.

[31] D. M. Martin, Jr. Iowa State University. Personal communication.

[32] C. A. Mead and L. A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980.

[33] M. L. Minsky. A LISP garbage collection algorithm using serial secondary storage, October 1963. Memo 58, Artificial Intelligence Laboratory, Massachussetts Institute of Technology, Cambridge, MA.

[34] Motorola, Inc. *MC88200 Cache/Memory Management Unit User's Manual.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[35] K. G. Muller. *On the feasibility of concurrent garbage collection.* PhD thesis, Tech. Hogeschool Delft, The Netherlands, 1976.

[36] K. Nilsen. Garbage collection of strings and linked data structures in real time. *Software—Practice and Experience* **18** (1988): 613–640.

[37] K. Nilsen. High-level goal-directed concurrent processing in Icon. *Software—Practice and Experience* **20** (1990): 1273–1290.

[38] K. Nilsen. A stream data type that supports goal-directed pattern matching on unbounded sequences of values. *Journal of Computer Languages* **15** (1990): 41–54.

[39] K. Nilsen and W. J. Schmidt. Cost-effective object-space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*. Accepted pending revision.

[40] K. Nilsen and W. J. Schmidt. Hardware-assisted general-purpose garbage collection for hard real-time systems. Technical Report 92–15, Department of Computer Science, Iowa State University, 1992.

[41] K. Nilsen and W. J. Schmidt. Preferred embodiment of a hardware-assisted garbage-collecting memory module. Technical Report 92–17, Department of Computer Science, Iowa State University, 1992.

[42] D. A. Pucknell and K. Eshragian. *Basic VLSI Design: Systems and Circuits*. Prentice Hall, New York, 1988.

[43] W. J. Schmidt and K. Nilsen. Architectural support for garbage-collected memory in hard real-time systems. Technical Report 91–23, Department of Computer Science, Iowa State University, 1991.

[44] W. J. Schmidt and K. Nilsen. Experimental measurements of a real-time garbage collection architecture. Technical Report 92–26, Department of Computer Science, Iowa State University, 1992.

[45] T. P. Singh. Hardware design of a real-time copying garbage collection system. Master's thesis, Iowa State University, 1990.

[46] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1990. Version 1.37.1. Available by anonymous `ftp` from `prep.ai.mit.edu`.

[47] J. A. Stankovic. Real-time computing systems: The next generation. In J. A. Stankovic and K. Ramamritham, editors, *Tutorial: Hard Real-Time Systems*, 14–37. Computer Society Press of the IEEE, 1988.

[48] S. M. Stapleton. Real-time garbage collection for general-purpose languages. Master's thesis, Iowa State University, 1990.

[49] G. L. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM* **18** (1975): 495–508.

[50] G. L. Steele, Jr. Private communication to H. G. Baker, Jr., March 1977.

[51] H. Taub and D. Schilling. *Digital Integrated Electronics*. McGraw-Hill, New York, 1977.

[52] S. S. Thakkar. Performance of Symmetry multiprocessor system. In M. Dubois and S. S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, 53–82. Kluwer Academic Publishers, Boston, 1990.

[53] M. D. Tiemann. *User's Guide to GNU C++*. Free Software Foundation, 1990. Version 1.37.1. Available by anonymous `ftp` from `prep.ai.mit.edu`.

[54] D. M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984, 157–167. Also distributed as *ACM SIGPLAN Notices* **19**(5): 157–167, May, 1987.

[55] P. L. Wadler. Analysis of an algorithm for real-time garbage collection. *Communications of the ACM* **19** (1976): 491–500.

[56] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, Reading, MA, 1985.

[57] J. L. White. Usenet communication, November 18, 1990.

[58] P. R. Wilson. Caching considerations for generational garbage collection. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, 1992, 32–42.

[59] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990, 87–98.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR 92-25
Submission Date: August 24, 1992